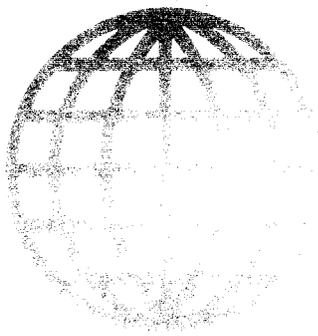


The COMPUTER JOURNAL®

Programming for Applications

Issue Number 4 4 May / June 1990 $3.95

Animation with Turbo C

Multitasking in Forth

Mysteries of PC Floppy Disks Revealed

DosDisk

Advanced CP/M

Real Computing

Forth Column

The Z-System Corner

The Computer Corner

ISSN # 0748-9331

The Computer Journal

Editor/Publisher
Art Carlson

Circulation
Donna Carlson

Contributing Editors
Bill Kibler
Bridger Mitchell
Clem Pepper
Richard Rodman
Jay Sage
Dave Wenstein

The Computer Journal is published six times a year by Technology Resources, 190 Sullivan Crossroad, Columbia Falls, MT 59912
(406) 257-9119

Entire contents copyright © 1990 by Technology Resources.

Subscription rates—$18 one year (6 issues), or $32 two years (12 issues) in the U.S., $24 one year surface in other countries. Inquire for air rates. All funds must be in U.S. dollars on a U.S. bank.

Send subscription, renewals, address changes, or advertising inquires to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, MT 59912, phone (406) 257-9119.

Registered Trademarks

It is easy to get in the habit of using company trademarks as generic terms, but these trademarks are the property of the respective companies. It is important to acknowledge these trademarks as their property to avoid their losing the rights and the term becoming public property. The following frequently used trademarks are acknowledged, and we apologize for any we have overlooked.

Apple II, II+, IIc, IIe, Lisa, Macintosh, DOS 3.3, ProDos; Apple Computer Company. CP/M, DDT, ASM, STAT, PIP; Digital Research. DateStamper, BackGrounder II, Dos Disk; Plu*Perfect Systems. Clipper, Nantucket; Nantucket, Inc. dBase, dBASE II, dBASE III, dBASE III Plus, dBASE IV; Ashton-Tate, Inc. MBASIC, MS-DOS, Windows, Word; MicroSoft. WordStar; MicroPro International. IBM-PC, XT, and AT, PC-DOS; IBM Corporation. Z80, Z280; Zilog Corporation. Turbo Pascal, Turbo C, Paradox; Borland International. HD64180; Hitachi America, Ltd. SB180; Micromint, Inc.

Where these and other terms are used in The Computer Journal, they are acknowledged to be the property of the respective companies even if not specifically acknowledged in each occurrence.

The COMPUTER JOURNAL

Issue Number 44 May / June 1990

Editorial 2

Animation with Turbo C 4
Part one in this issue covers the basic tools and the Turbo C Ver. 2.0 graphics libraries.
By Clem Pepper.

Multitasking in Forth 11
Implementing multitasking with New Micros F68FC11 and Max-Forth.
By Matthew Mercaldo.

Mysteries of PC Floppy Disks Revealed 16
Here's the information you need to understand FM, MFM, and the twisted cable.
By Richard Rodman.

DosDisk 19
The MS-DOS disk format emulator for CP/M systems.
By Daniel J. Mareck.

Advanced CP/M 21
ZMATE Z-System programmer's editor and using lookup and dispatch for passing parameters.
By Bridger Mitchell.

Real Computing 25
The NS32000.
By Richard Rodman.

Forth Column 27
Forth news, and handling strings in Forth.
By Dave Weinstein.

The Z-System Corner 29
Working with the MEX telecommunications package.
By Jay Sage.

The Computer Corner 40
By Bill Kibler.

Editor's Page

Challenge of the Future Part 2

My comments in issue #43 on the future employment prospects for computer programmers were intended to incite response and violent disagreement. I have not received any reaction from our readers, but instead have noted several reinforcing articles in other publications.

The UNIX Journal (7620 242nd Street S.W., Edmonds, WA 98020-5463) carried an article entitled *Software Development in the 1990s* by Loren West. In this article, West discusses several challenges which face the software industry in the next decade. Portions of the article are: ....writing software is a clerical task, much more suited for computers, ...envision software production in the '90s being completely different from current methods. The programmer will use his CASE tool to describe the application...., ...the manufacturing plant will custom manufacture the software for the customer. Application programmers will become obsolete, and they will be moved to less clerical positions...., ...designing software, and virtually no time programming.

An article on *EEs' new challenge: electronic immigrants* by Robert Bellinger in the March 12, 1990 issue of *Electronic Engineering Times* is subtitled *As telcom advances, work may gravitate to Third World EEs.* Portions of the article read: ....five or 10 years down the road, a Pakistani or Indian EE might be doing the job you're working on now...., getting more profitable to automate or transmit work to areas that offer low labor rates. ....Texas Instruments, Inc., Hewlett-Packard Co. and Digital Equipment Corp. are farming software applications to Indian subsidiaries. ....using phone lines, faxes and Federal Express to relay work to UNIX programmers based in Calcutta of Bangalore.

West's article talks about programming being turned over to automated systems, and Bellinger talks about the work being done overseas. Consider the prospects of competing with a software engineer with three years of Unix programming experience and a master's degree who earns $7,000 a year and is equipped with a Sun workstation. Bellinger calls them "electronic immigrants" because they can remain in their country and communicate at speeds predicted to reach 1.5 Mbit by 1995 and 45 Mbit by 2000.

American industry is being financially squeezed. They are downsizing and are searching for non-traditional methods of cutting costs and reducing long term employment and pension commitments —automation and transmission of work to low wage areas is very attractive to them. Much of the routine data entry and word processing work is already being sent overseas and the foreign placement of more advanced programming is accelerating with 1995 projected as the point at which the effect will be apparent.

Current cuts in defense spending are another factor affecting the technical job market, and many of the employment losses will be permanent because the companies will switch to automation and/or overseas sources instead of refilling positions which have been eliminated.

The crystal ball is too cloudy to accurately predict the future in our fast changing environment—we will be fortunate if we can foresee some of the major trends. I feel that five years from now most job opportunities in large organizations in established fields will be for high level people with a lot of experience who can supervise and coordinate foreign nationals working in their own countries. There will also be a need for advanced technician grade people to service and maintain the complex devices in offices and factories.

That's for the established fields. An entirely different situation will exist for entrepreneurial concerns working on new developments where fully automated and foreign sources do not yet exist. These pioneers will require high tech people with extensive experience in abstruse specialities, who also know how to program.

Employment positions for people defined as programmers will essentially disappear during the next five years. Everyone will be expected to be able to use the automated CASE program generation tools, or to do the necessary programming as part of their primary job function. Recruiters will be searching for engineers and designers who can also program, instead of people who just program what someone else has designed. In the past, programming was a speciality performed by a very small aristocracy. In the future, hiring someone who just programs would be like hiring someone to do simple multiplication and division for engineers who could not understand how to use a calculator.

Current employment ads already reflect this trend. Some excerpts are: ...experience in instrumentation, control systems, data acquisition and microprocessors. Experience with electronic CAD software a plus. ...design, development, test & integration of software in a high order language to be used in automated instrument test equipment. ...to continue the development of fuel controlling devices, point-of-sale consoles, and financial network communications. ...experience in software development with real time embedded controllers. ...analog background in electronic circuits and devices. Experience with analog simulation. ...networking basics including mixed networks. ...strength in computer-aided software engineering.

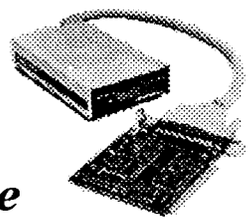
It is time that we reduce the emphasis on computers and programming as entities in themselves, and pay more attention to using software and hardware to accomplish tasks. Software and hardware design are still required, but they are part of another discipline rather than a separate function. We are researching this changing market and will be presenting more application oriented articles to help prepare you for the future. Material on embedded controllers, serial and parallel communications, interfacing, digital signal processing, data acquisition and analysis, motor control, and parallel processing is in preparation. We look forward to your suggestions on additional topics—and your articles!

(Continued on page 38)

The Best External 3.5" Drive Kit

$269

- Works with both 720KB and 1.4MB diskettes in any PC, XT or AT
- Automatically senses whether you are using 720KB or 1.4 MB diskettes
- All Inclusive: drive, cable, card and software included
- Simple installation
- Coexists with existing drive controllers
- 1 year warranty



Cat. #MEGM $269 **megamate**

MegaMate 2.8MB kit also available. MegaMate II allows you to read/write/format 720K, 1.4MB & 2.8MB disks. Includes all features of the standard MegaMate. **Cat. #MEGM2, $399.**

snOOp II The Intelligent Disassembler

$49

Look What's Inside

snOOp II takes incomprehensible object code and turns it into assembly language source code. That's not so unique, but **snOOp II** is an intelligent disassembler that automatically comments each line of code, and labels jump targets. It will even explain what each instruction does in the context of the program. That's built in help that borders on an expert system (actually more like a tutor, ready to explain everything).

SNOOP 80 x 86 Disassembler

Help	F1
Enter File Name	F2
Disassemble file	F3
Output .ASM File	F4
Output .MAP File	F5
Automatic Disassembly	F6
Automatic Options	F7
Tutorial	F8
RPN Caluculator	F9
Quit	F10
F1 to F10	

snOOp II also has one of the best tutorials we've ever seen. It disassembles a file that is especially commented to explain it's workings to you. As you progress through the file you learn how to identify data areas **snOOp** couldn't catch, how to interpret what you see, and how to move quickly and smoothly through software — tracing the logic automatically as you go.

snOOp correctly interprets all instructions for the 8086, 8088, 80186, 80286, 80386, 8087 and 80287, 80387 processors and coprocessors. It identifies interrupt calls and port addresses.

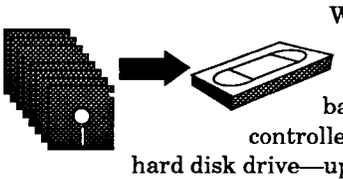
So what's this good for? Well it's certainly the best way around to learn assembly language programming. And it's the best way to make changes to software when you don't have the source code.

Cat. #SNOO, $49 Order today and save!!

World's Most Reliable Backup

$199

Retail: $399



Why hassle with floppy disk backup programs when you already own half of the most reliable tape backup system? The Videotrax controller board allows you to backup your hard disk drive—up to 240 megabytes—on a single, inexpensive videotape using almost any VCR.

World's Most Reliable Backup

Videotrax is made by AlphaMicro, a company which has been creating and perfecting videotape backup technology for nearly 10 years. Using error checking codes and multiple copies of your data, Videotrax makes a standard grade videotape more reliable than any other backup system.

- Store up to 240 MB on a single videotape
- Use any VHS, Beta, 8mm, or European VCR with video input and output jacks
- Time-delayed unattended backup
- File-by-file backup for all, single, or groups of files
- More reliable than streamer tape, floppies, or hard disks
- Stores 1MB per 1.3 minutes
- 30 day money back guarantee

At only $199, Videotrax is the most cost-effective backup system. Order yours today. **Cat. # VIDT.**

IBM PS/2 (Microchannel) Version only $249, **Cat. #VIDM**

We guarantee that you'll be completely satisfied with your purchase or else:

- We will replace the products ***OR***
- We will promptly and cheerfully refund your money

Please call Customer Service at (805) 524-4189 for a return authorization number before returning any products. Guarantee is limited to the first 30 days after purchase.

Epson/Okidata Upgrade

This easy-to-use enhancement makes your Okidata ML82A, 83A, Epson MX, FX, RX, or JX a powerful new printer. Don't toss out your workhorse Epson or Okidata when you can easily install this hardware upgrade and give it new life.

Get the latest IBM Graphics Printer capabilities, including the full IBM Character Set, 10, 12, & 17 pitch text, IBM/Epson-compatible graphics (except on RX), and super smart looking near letter quality. You get all these features via software control or by simply touching the three buttons on your control panel. Does everything Okigraph, Plug-n-Play and Graftrax do and more! Comes complete with manual. Satisfaction guaranteed. Order today.

For Epson printers, order **Dots-Perfect, Cat. #DOTP, $69.**

For Okidata printers, order **PC-Writer, Cat. #PCWR, $79.**

CENTRAL COMPUTER PRODUCTS

Serving Computer Users Since 1982

Call for a PC or CP/M Catalog

1•800•456•4123

Ad Code: 110-6. Add $5.50 shipping. CA & NY Res. Add Sales Tax.

330 Central Avenue • Fillmore, CA 93015 • FAX (805) 524-4026

Animation with Turbo C Ver. 2.0

Part 1: The Basic Tools

by Clem Pepper

My last article described many features of Borland's Turbo C graphics library. This article, picking up where the previous left off, emphasizes application of the library to screen animation. As a point of interest, a check of the indexes of the Borland User and Reference Guides reveals a total lack of any reference to animation. As our experience develops, the reason for the omission becomes clear. With respect to animation the library leaves much to be desired in ease of use, in speed, and in overall performance. Still it is not all bad. I have a rather elegant war game running, which, while somewhat less than marketable in quality was a challenge to program and is fun to play. Most of what you need to know for creating your own screen action will be found in this writing. Part 2 will continue with the development of a full fledged action game.

A few reminders before plunging into the awesome details. Three essentials of programming—planning, patience and persistence--are a must. Think out your objectives. Make sketches of your game figures using grids similar to those shown previously. Work up the construction statements on scribble paper before typing them into your program. And speaking of typing, a good editor really speeds up the job. Attributes of a good editor are speed and ease of copy and move operations.

Control logic in game programs becomes complex very rapidly, in particular when several game objects interact with each other. While mistakes in logic do impede progress, many errors I contend with arise from silly mistakes—missing ('s, ;'s, {'s and the like. A major source is shooting from the hip in an effort to resolve problems quickly instead of taking a break to sit back and think out solutions. And I have even learned it pays to go back to the book from time to time.

There is a caution I must give before going further. The Turbo C graphics bypass the BIOS ROM for direct interaction with the hardware. This, I would imagine, assumes full IBM compatibility. The possibility of adverse effects with a computer that is less than fully compatible may exist. My own Zenith is described as 99 percent compatible. One program I experimented with ran perfectly. But in some mysterious way it interacted with the system tracks on my hard disk. As a result I had to re-format the drive and write back all the files. That program, I assure you, is NOT included with this article. All the examples perform properly on my computer. I have a CGA adapter. All my program work is done on a floppy in drive A:, I keep no data files on my hard drive.

Well, so much for philosophy. Let's animate.

Entering the Graphics Mode.

Listing 1, HEADING.C, should be employed with each of the example listings provided. The graphic declarations and functions are a requirement for any program making use of the graphics library. If you have other than a CGA adapter you should make an appropriate correction.

One observation. The declaration

```
int graphdriver = CGA; /* or EGA or other adapter */
```

can be written as

```
int graphdriver = DETECT;
```

in which case the function initgraph() will check your system for the adapter in use and load the appropriate graphics driver. If I were putting out a program in the public domain or as shareware I would use DETECT. For my own use I prefer specifying the adapter I have.

Also note that once initgraph() has been called graphmode cannot be changed.

```
/* HEADING.C
** Include this listing with each of the example programs.
** Note that this listing includes main() {
** Insert program #includes, graphmode and needed
** declarations with each of the example programs.
*/

/***** insert #includes<> *****/

/* == Begin program == */
main()
{
int graphdriver = CGA;       /* graphics driver       */
int graphmode = ?;           /* specify 0, 1, 2, or 3 */
  /* ** graphmode cannot be changed later in a program ** */
int errorcode;               /* graphics error code   */

 /*****  insert program declarations *****?

initgraph(&graphdriver, &graphmode, "e:\\btc20");
  /* ** replace "e:\\btc20" with your directory location ** */
errorcode = (graphresult());      /* get result code */

  /* ** graphics error function routine call ** */
   if (errorcode != grOk)   /* always check for error */
  {
    printf("Graphics error: %s\n",grapherrormsg(errorcode));
    exit(1);
   }

  /* ** call to set background color ** */
  setbkcolor(BLUE);


Listing 1.  Heading to be used with the example programs.
```

```
/* ARMY_TNK.C
** Left/right directed stick line tank figure.
** using the TURBO C Ver. 2.0 library routines.
** Tank gun omitted to permit multi-angle shell firing.
*/
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <ctype.h>
#include <graphics.h>

/* ** global declarations ** */
#define SFKEY 0x16
int asc, scn;

/***** insert heading.c *****/

/* ** additional declarations in main() ** */
 *int tdir_flg = 0;           /* tank moves to left    */
int i = 1, left_col = 300, rite_col = 319;
char run = '0';

  /* ** display exit, tank move messages ** */
   outtextxy(20,20,"Arrow keys move, stop the tank.");
   outtextxy(20,35,"Pressing 'q' exits the loop.");
    moveto(20,50);
     outtext("Then press any key to quit.");

  /* ** begin animation ** */
  /* ** define viewport ** */
   while(i) {
     if(tdir_flg == 1)         {
      left_col += 5; rite_col += 5;
       if(left_col >= 300) tdir_flg = 0;
                                }
     else if(tdir_flg == 0)      {
      left_col -= 5; rite_col -= 5;
       if(left_col <= 5) tdir_flg = 1;
                                 }

    setviewport(left_col,189,rite_col,199,1);
     draw_tank();
      delay(30);  clearviewport();
       if(kbhit() == 0)  continue;
       else              {
            rd_nonasky(); {
  /* ** move tank to the left ** */
         if(asc == 75) { tdir_flg = 0; continue; }
  /* ** move tank to the right ** */
    else if(asc == 77) { tdir_flg = 1; continue; }
  /* ** halt tank in current position ** */
    else if(asc == 72) { tdir_flg = 2; continue; }
                       }
        run = (toupper(toascii(scn)));
          if(run == 'Q')  i = 0;
                          }
                }
     draw_tank();   /* for stationery view */
      getch();      /*  before exiting.    */
       closegraph();  /* restore text mode */
}

/* == draw tank as sequence of horiz lines == */
draw_tank()
{
  setlinestyle(0,0,1);  /* solid line, one pixel wide */
  setcolor(2);          /* tank top is red */
   line(9,0,15,0);      /* segment a */
  setlinestyle(0,0,3);  /* solid line, 3 pixels wide */
   line(7,2,17,2);      /* segment c */
  setlinestyle(0,0,1);  /* solid line, one pixel wide */
   line(9,4,15,4);      /* segment d */
  setcolor(1);          /* lower tank is green */
   line(5,5,19,5);      /* segment e */
   line(4,6,19,6);      /* segment f */
   line(5,7,19,7);      /* segment g */
   line(6,8,18,8);      /* segment h */
   line(7,9,17,9);      /* segment i */
}

/* == read non-ascii key == */
int rd_nonasky()
{
union REGS regs;
 regs.h.ah = 0;
 regs.h.al = 0;
  int86(SFKEY, &regs, &regs);
   asc = regs.h.ah; /* ASCII code */
   scn = regs.h.al; /* SCAN code  */
}
```

Listing 2. A program illustrating animation using setviewport().

Animation With Viewports

Because viewports were discussed last time with program examples we have already made their acquaintance. We continue here with applications leading to screen action in which more than one viewport is active.

Listing 2, ARMY_TNK.C, is a repeat of TANK.C with additional features. These are provisions for controlling the tank's motion across the screen. It can be directed left, right, or motion stopped by using the left, right, and up arrow keys. The program also includes a text message. The display of text is covered in a later section.

In this program we employ the function draw_tank() on each pass through the while() loop in main(). The program employs direction flags set by arrow keys to control tank movement from side to side and to halt the motion. Note that the while loop opens with if logic for reading the flags. The column values are incremented, decremented, or left unchanged depending on the value of the flag. A test for the screen edge is also included. If the tank is at an edge the direction is reversed by changing the flag value. We will find the use of flags especially helpful as we expand the number of objects in our animation.

Pressing the 'q' key terminates the action. Because both ASCII and non-ASCII keyboard input is employed we need an approach that does not require double keying to read the input. The function, rd_non_asky() provides a solution. Through this function we can read any key that is pressed.

The library function kbhit() tests the keyboard for input. If none is present its value is zero and the loop continues. If a key has been pressed the function rd_nonasky() is called. This function, described in an earlier article, can be employed to read any keypress, ASCII or non-ASCII as follows:

```
int asc, scn;     /* global declaration */
int rd_non_asky()
{
union REGS regs;
  regs.h.ah = 0;
  regs.h.al = 0;
   int86(0x16,&regs,&regs);
     asc = regs.h.ah; /* ASCII code */
     scn = regs.h.al; /* SCAN code */
}
```

The comments apply to the response to a non-ASCII key, which are the function and keypad keys and the other special purpose keys. For these keys the scan code is zero. However if an ASCII key is pressed its integer value of the ASCII character is in the scan code variable, (scn). A statement such as

```
char run = toupper(toascii(scn)));
```

sets the character variable run to that keyed in. This eliminates the need for getch() to detect an ASCII keypress.

```
/* BOMR.C
** Construction of a three plane, left-to-right
** aircraft group using setviewport().
** Turbo C Version 2.0 graphics.
*/
#include <stdio.h>
#include <graphics.h>

/***** insert heading.c *****/

/* ** additional declarations in main() ** */
int i = 54, left_col = 10, rite_col = 30;


  /* ** display exit message ** */
   outtextxy(20,120,"Press any key to exit.");

  /* ** begin animation ** */
  /* ** define viewport ** */
   while(i--) {

  /* ** draw first bomber ** */
   setviewport(left_col,0,rite_col,20,1);
    draw_bomber();
  /* ** draw second bomber ** */
   setviewport(left_col-10,20,rite_col-10,40,1);
    draw_bomber();
  /* ** draw third bomber ** */
   setviewport(left_col+15,25,rite_col+15,45,1);
    draw_bomber();

  /* ** erase bombers ** */
      delay(20);
   setviewport(left_col,0,rite_col,20,1);
     clearviewport();
   setviewport(left_col-10,20,rite_col-10,40,1);
     clearviewport();
   setviewport(left_col+15,25,rite_col+15,45,1);
      clearviewport();
        left_col += 5; rite_col += 5;
               }
  /* ** final draw for viewing ** */
   setviewport(left_col,0,rite_col,20,1);
    draw_bomber();
   setviewport(left_col-10,20,rite_col-10,40,1);
    draw_bomber();
   setviewport(left_col+15,25,rite_col+15,45,1);
    draw_bomber();

      getch();
       closegraph();
}

/* == draw bomber as sequence of h/v lines == */
draw_bomber()
{
 setlinestyle(0,0,1);  /* solid line, one pixel wide */
  /* ** draw wings ** */
    setcolor(1);        /* wings, tail are green */
     line(9,1,9,8);     /* segment w1 */
     line(10,2,10,8);   /* segment w2 */
     line(11,3,11,8);   /* segment w3 */
     line(12,4,12,8);   /* segment w4 */
     line(13,5,13,8);   /* segment w5 */
     line(14,6,14,8);   /* segment w6 */
     line(14,12,14,14); /* segment w7 */
     line(13,12,13,15); /* segment w8 */
     line(12,12,12,16); /* segment w9 */
     line(11,12,11,17); /* segment w10 */
     line(10,12,10,18); /* segment w11 */
     line(9,12,9,19);   /* segment w12 */

  /* ** draw tail ** */
     line(1,5,1,15);   /* segment s1 */
     line(2,5,2,15);   /* segment s2 */
     line(3,6,3,14);   /* segment s3 */
     line(4,7,4,13);   /* segment s4 */

  /* ** draw fuselage ** */
    setcolor(2);       /* fuselage is red */
     line(8,8,15,8);   /* segment f1 */
     line(4,9,18,9);   /* segment f2 */
     line(0,10,20,10); /* segment f3 */
     line(4,11,18,11); /* segment f4 */
     line(8,12,15,12); /* segment f5 */
}
```

Listing 3. A program illustrating animation of multiple objects using setviewport().

The screen coordinates are defined in the first statement of the loop: a call to library function setviewport(). A delay follows the tank display to allow time for viewing. The viewport is then cleared with the function call clearviewport(). An update of the screen coordinates follow, after which the loop repeats.

Two conditions impose limits to the utility of the viewport for animation. The first, and least serious, is that a new figure has to be constructed on each pass through the loop. For a small object, such as the tank, this is not too terrible. The second constraint, however, is a tricky one to deal with.

Note that clearviewport() has no coordinates. The function prototype is "void clearviewport(void)." A little experimenting shows us that only one viewport, the most recent, will be cleared by this function call. So how do we display two or more animated objects at the same time?

Listing 3, BOMR.C, illustrates a solution. In this program an airplane, defined as a bomber, is created in a function call for display in a viewport. Construction details are shown in Figure 1. This one play object is then replicated twice to form a flight group of three aircraft. The screen display, by the way, is more impressive than might appear from the sketch.

The replication is formed by simply modifying coordinates in the function

```
setviewport(int left,int top,int right,int bottom,clip);
```

The coordinates for the second and third port locations are offsets from the first. We don't have to do this, but there are benefits to doing so when we get into interaction with objects on the screen. The fewer variables to keep track of, the better.

The while loop operation begins by displaying the three aircraft in their flight formation. There follows a short delay for viewing. Then each viewport, beginning with the first, is set again, after which it is cleared.

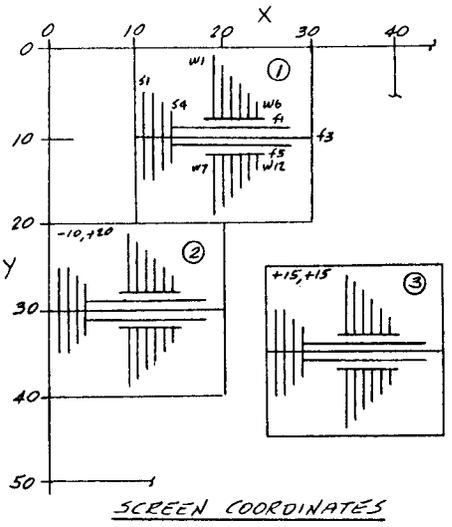
Not only is this inefficient, it slows down the action in two respects. First, the figure must be re-drawn from scratch on each setviewport() call, and second, we have to make the call twice.

The viewport does have its place, but there is another, generally better, approach. We'll discuss it now.

Animation with putimage()

This function was not discussed last time. In Listing 4, HELI.C,the play object represents a helicopter that we will move around much like a hummingbird at a sugar water feeder. Construction details are provided in Figure 2.

In contrast to the viewport approach the object is constructed ahead of any animation and saved in a buffer. This speeds up the screen action to a significant degree. The listing includes four new declarations to be added to those given in the heading file:




```
setlinestyle(0,0,1);  /* solid line, one pixel wide */
 /* ** draw wings ** */
   setcolor(1);          /* wings, tail are green */
    line(9,1,9,8);       /* segment w1 */
    line(10,2,10,8);     /* segment w2 */
    line(11,3,11,8);     /* segment w3 */
    line(12,4,12,8);     /* segment w4 */
    line(13,5,13,8);     /* segment w5 */
    line(14,6,14,8);     /* segment w6 */
    line(14,12,14,14); /* segment w7 */
    line(13,12,13,15); /* segment w8 */
    line(12,12,12,16); /* segment w9 */
    line(11,12,11,17); /* segment w10 */
    line(10,12,10,18); /* segment w11 */
    line(9,12,9,19);     /* segment w12 */

  /* ** draw tail ** */
     line(1,5,1,15);     /* segment s1 */
     line(2,5,2,15);     /* segment s2 */
     line(3,6,3,14);     /* segment s3 */
     line(4,7,4,13);     /* segment s4 */

  /* ** draw fuselage ** */
    setcolor(2);         /* fuselage is red */
     line(8,8,15,8);     /* segment f1 */
     line(4,9,18,9);     /* segment f2 */
     line(0,10,20,10); /* segment f3 */
     line(4,11,18,11); /* segment f4 */
     line(8,12,15,12); /* segment f5 */

   Line assignments for a single aircraft

   Offsets for aircrafts two and three based on
   starting location of x = 10, y = 0 for
   aircraft one:
     Aircraft two..... -10, +20
     Aircraft three .. +15, +25
```

Figure 1. Construction details for flight group of three aircraft.

```
char far*heli;
char buffer[80];
short x,y;
unsigned numbytes;
```

As with the army tank, the helicopter is constructed from short segments of straight lines. The construction follows the same pattern as for the tank, that is, set the linestyle, set the color, and draw the lines. (Although the examples in this article use straight line constructions we can also use filled circles, polygons, ellipses and so forth when appropriate.)

With the figure construction defined, our next efforts are to save it in a buffer. The first step is to determine the size of the

```
/* HELI.C
** A rudimentry graphics program using putimage() for
** animation.
*/

#include <stdio.h>
#include <alloc.h>
#include <graphics.h>

/***** insert heading.c *****/

/***** additional declarations in main() *****/
char far *heli;
char buffer[80];
short x, y;
unsigned numbytes, ch = 0;

  /* ** draw initial helicopter figure ** */
    setlinestyle(0,0,1);  /* solid line, one pixel wide */
    setcolor(3);          /* heli is brown */
     moveto(4,0);  lineto(5,0); /* segment a */
     moveto(0,1);  lineto(9,1); /* segment b */
     moveto(4,2);  lineto(5,2); /* segment c */
     moveto(3,3);  lineto(6,3); /* segment d */
     moveto(3,4);  lineto(6,4); /* segment e */
     moveto(3,5);  lineto(6,5); /* segment f */
     moveto(2,6);  lineto(7,6); /* segment g */
     moveto(1,7);  lineto(2,7); /* segment h1 */
     moveto(7,7);  lineto(8,7); /* segment h2 */
     moveto(1,8);  lineto(2,8); /* segment i1 */
     moveto(7,8);  lineto(8,8); /* segment i2 */

  /* ** determine storage needed ** */
     numbytes = imagesize(0,0,15,15);

  /* ** allocate buffer ** */
     if((heli = (char far *) malloc(numbytes))
        == (char *)NULL)
   {
     closegraph();
     printf("Not enough memory for storage.");
     exit(0);
   }
     getimage(0,0,15,15,heli); /* save the image */
     clearviewport(); /* clear screen of saved image */


    cleardevice();  setcolor(3);
    outtextxy(10,10,"Demo using putimage");
      x = getmaxx()/2;  y = getmaxy()/2;
      putimage(x,y,heli,XOR_PUT);
      outtextxy(10,getmaxy()-50,
       "q=exit,h=left,j=down,k=up,l=right");

  /* ** perform animation ** */
    while(ch != 'q')  {
     ch = getch();

  /* ** 1st erase at last position ** */
    putimage(x,y,heli,XOR_PUT);
    switch(ch)  {
      case 'h': x -= 2; break; /* 2 pixels left */
      case 'l': x += 2; break; /* 2 pixels rite */
      case 'j': y += 2; break; /* 2 pixels down */
      case 'k': y -= 2; break; /* 2 pixels up */
                      }
  /* ** redraw at new position ** */
    putimage(x,y,heli,XOR_PUT);
                        }
  /* ** restore original mode ** */
    closegraph();
}
```

Listing 4. Program illustrating simple animation using putimage().

```
/* BOMBER.C
** Construction of a three plane, left-to-right
** aircraft group using the plane of planes2.c.
** Development program for BOMR.C
*/
#include <stdio.h>
#include <alloc.h>
#include <graphics.h>

/***** global declarations *****/
int leftbd_col = 10,leftbd_top = 0, riteb_col = 30;
int leftbe1_top, leftbe2_top, leftbe3_top;
int leftbe1_col, leftbe2_col, leftbe3_col;
char far *bomr;

/***** insert heading.c *****/

/* ** additional declarations in main ** */
int i = 20;
char run = ' ';
unsigned numbytes;
char buffer[80];

  /* ** draw initial bomber figure ** */
   setlinestyle(0,0,1);  /* solid line, one pixel wide */
    setcolor(2);          /* fuselage is red */
  /* ** draw wings and tail ** */
    setcolor(1);          /* wings, tail are green */
     moveto(9,1);    lineto(9,8);   /* segment w1 */
     moveto(10,2);   lineto(10,8);  /* segment w2 */
     moveto(11,3);   lineto(11,8);  /* segment w3 */
     moveto(12,4);   lineto(12,8);  /* segment w4 */
     moveto(13,5);   lineto(13,8);  /* segment w5 */
     moveto(14,6);   lineto(14,8);  /* segment w6 */
     moveto(14,12);  lineto(14,14); /* segment w7 */
     moveto(13,12);  lineto(13,15); /* segment w8 */
     moveto(12,12);  lineto(12,16); /* segment w9 */
     moveto(11,12);  lineto(11,17); /* segment w10 */
     moveto(10,12);  lineto(10,18); /* segment w11 */
     moveto(9,12);   lineto(9,19);  /* segment w12 */
     moveto(1,5);    lineto(1,15);  /* segment s1 */
     moveto(2,5);    lineto(2,15);  /* segment s2 */
     moveto(3,6);    lineto(3,14);  /* segment s3 */
     moveto(4,7);    lineto(4,13);  /* segment s4 */

  /* ** draw fuselage ** */
    setcolor(2);          /* fuselage is red */
     moveto(8,8);  lineto(15,8); /* segment f1 */
     moveto(4,9);  lineto(18,9); /* segment f2 */
     moveto(0,10);  lineto(20,10); /* segment f3 */
     moveto(4,11);  lineto(18,11); /* segment f4 */
     moveto(8,12);  lineto(15,12); /* segment f5 */

  /* ** determine storage needed ** */
    numbytes = (unsigned int)imagesize(0,0,20,20);

  /* ** allocate buffer ** */
    bomr = (char *)malloc(numbytes);

     getimage(0,0,20,20,bomr); /* save the image */
      cleardevice();

   /* ** display exit text on screen ** */
     outtextxy(50,160,"Press any key to exit");

   /* ** begin animation ** */
    do  {

   /* ** draw first bomber ** */
     leftbe1_col = leftbd_col; leftbe1_top = leftbd_top;
      draw_bomber();

   /* ** draw second bomber ** */
     leftbd_col -= 10;  leftbd_top += 20; riteb_col -= 10;
     leftbe2_col = leftbd_col; leftbe2_top = leftbd_top;
       draw_bomber();

   /* ** draw third bomber ** */
     leftbd_col += 25;  leftbd_top += 5; riteb_col += 25;
     leftbe3_col = leftbd_col; leftbe3_top = leftbd_top;
      draw_bomber();
       delay(40);
        clr_bomrs();

   /* ** advance to next position ** */
     riteb_col -= 5; leftbd_col -= 5; leftbd_top = 0;

     if(riteb_col >= 299)  {
       leftbd_col = 10, riteb_col = 30;
                          }
       if(kbhit() == 0) continue;
        run = (toupper(getch()));
         if(run == 'Q') i = 0;
                } while(i--);
       getch();  closegraph();
}

/* == erase bombers == */
clr_bomrs()
{
  /* ** erase first bomber ** */
    putimage(leftbe1_col,leftbe1_top,bomr,XOR_PUT);
     delay(10);

  /* ** erase second bomber ** */
     putimage(leftbe2_col,leftbe2_top,bomr,XOR_PUT);
      delay(10);

  /* ** erase third bomber ** */
     putimage(leftbe3_col,leftbe3_top,bomr,XOR_PUT);
      delay(10);
}

/* == draw bomber from saved image data == */
draw_bomber()
{
  putimage(leftbd_col,leftbd_top,bomr,COPY_PUT);
 }
```

Listing 5. Program illustrating animation with multiple calls to putimage().

buffer. This is performed with the statement

```
numbytes = imagesize(0,0,10,10);
```

There is little danger of overflowing memory with such a diminutive object but for safety we hedge with

```
if(heli = (char *)malloc(numbytes) == (char *)NULL)
{
  closegraph(); /* return to text mode */
  printf("Not enough memory for storage.");
  exit(0);
}
```

Assuming all is well the next step is to save the image now residing in the buffer. I find the name of the saving function, getimage(), to be confusing. But that's the way it is.

```
getimage(0,0,10,10,heli); /* save the image */
```

and now comes the bad part:

```
clearviewport(); /* clear screen of the saved image */
```

The "bad" designation arises from a momentary flash that appears at the upper left corner of our screen. The flash from one image isn't all that bad, but wait until we have or six or eight.

```
setlinestyle(0,0,1);  /* solid line, one pixel wide */
 setcolor(3);          /* heli is brown */
  moveto(4,0);  lineto(5,0); /* segment a */
  moveto(0,1);  lineto(9,1); /* segment b */
  moveto(4,2);  lineto(5,2); /* segment c */
  moveto(3,3);  lineto(6,3); /* segment d */
  moveto(3,4);  lineto(6,4); /* segment e */
  moveto(3,5);  lineto(6,5); /* segment f */
  moveto(2,6);  lineto(7,6); /* segment g */
  moveto(1,7);  lineto(2,7); /* segment h1 */
  moveto(7,7);  lineto(8,7); /* segment h2 */
  moveto(1,8);  lineto(2,8); /* segment i1 */
  moveto(7,8);  lineto(8,8); /* segment i2 */
```

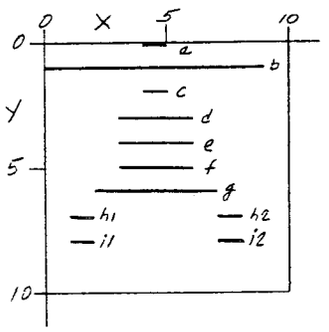


Figure 2. Helicopter construction from line segments.

The creation of each new object follows the exact same steps as just illustrated for "heli." Each additional object must be declared with a far pointer and the object name. Other than that no new declarations are required.

This program differs from ARMY_TNK in several respects. Instead of a free wheeling while loop we will position the heli ourselves from the keyboard. Instruction text for its control is displayed on the screen.

The on-screen display is accomplished by a call to putimage(). This function requires four arguments:

```
putimage(int left, int right, void far *image, int action);
```

```
/* SCORE.C
** Illustrating viewport text display with
** outtextxy() using the TURBO C Ver. 2.0
** library routines.
*/

#include <stdio.h>
#include <conio.h>
#include <graphics.h>

/***** global declarations *****/
int score = 0, score_flg = 1;
char score_buf[80];


/***** insert heading.c *****/

/***** additional declaration in main() *****/
int i = 20;

  /* ** begin animation ** */
  /* ** define viewport ** */
   while(i--) {
    if(score_flg == 1) /* flag is set if score updated */
     clearviewport();  /* clear for score update */
      setviewport(0,0,319,10,0); /* restore port */
       sprintf(score_buf,"You have %d points.",score);
      outtextxy(0,0,score_buf); /* display updated score */
     score_flg = 0;  score_up();
    delay(30);
                }
     moveto(20,50);  setcolor(2);
     outtext("Press any key to exit.");
  /* ** the message will not print with a 1 in ** */
  /* ** setviewport 0,0,319,10,0). Try it!     ** */
       getch();
        closegraph();
}

/* == update score == */
score_up()
{
    score += 100; score_flg = 1;
}
```

Listing 6. Illustrating use of sprintf() and outtextxy in a viewport for graphic programs requiring text

```
The putimage() function prototype:
void far putimage(int left, int top, void far *image_buffer, int action);


ACTION      WHAT IT DOES
------      ------------
COPY_PUT    Saved image is drawn at the specified screen location. An
            existing image is overwritten.

XOR_PUT     Each pixel of the saved image is exclusive-ORed with its
            respective pixel of the currently displayed image.  If an
            image exists, it is erased.  If an image does not exist it
            is drawn.

OR_PUT      Pixel values from the saved image are logically ORed with
            existing pixels in the area where an image is being drawn.

AND_PUT     An image is drawn by performing a logical AND of the existing
            pixel value with the corresponding pixel in the saved image.

NOT_PUT     Each pixel bit of the saved image is logically inverted.
            The inverted values are written over existing pixels in the area
            where an  image has been drawn.  Effect is to change its color.
```

Table 1. Application of the putimage() action constants to saved and on-screen images.

The two integers define the upper left corner of the location on screen, in SCREEN (not viewport!) coordinates. The action argument is one of five described in Table 1. The two we are concerned with in our animations are COPY_PUT and XOR_PUT. We can use either of these to display the object, then use XOR_PUT to erase it. XOR_PUT performs a pixel exclusive-OR which simply blends the object in with the background.

The use of outtextxy() will be discussed in the next section. In this routine we display a message beginning with column 10, row 10 that this is a demo of putimage. A second message near the screen bottom instructs us on how to control the heli and exit the program. In between we use putimage() to initially display the heli.

The while loop remains active until we press the 'q' key. The opening statement waits for a key press, then erases the object. Following a press of one of the four positioning keys the object is redrawn at its new position. The program then awaits a new key press before continuing on.

There are several good features of putimage(). The first is speed, it is quite fast. The second is we can erase it anytime with a subsequent call, which means we can maintain multiple images on screen. A third is that with the figure's construction saved in the buffer we can sprinkle the same object all over the place if we need to do so. And go back and erase them all in a timely manner.

There are some qualities not so good. The most obvious is the initial screen flash. The other most apparent arises when one of our objects intersects another. Without corrective effort our screen can quickly become littered with bits of garbage arising from the encounter. We will deal with this later when we look at multiple display objects.

Mixing calls to setviewport() with putimage() where both involve changing coordinates does not appear to work very well. If the initial image is created and displayed using getimage() and putimage() the first pass through will find the object correctly positioned on the screen. What I have observed then is the display of another object) using setviewport() and clearviewport() that appears to add offsets to the screen coordinate variables for the other object(s). They simply show up somewhere else on the screen. There is, none-the-less a needed application of setviewport() for the display of text. This is discussed in the next section.

Combining Text With Graphics

We cannot use printf() when in the graphics mode. The two library functions available for strings are outtext() and outtextxy(). These are identical except that the latter includes the column (x) and row (y) coordinates in the call.

The outtext() function prototype is

```
void far outtext(char far *textstring);
```

Outtext() displays a null terminated string, that is, a character array terminated in a null (\0), at the current screen position. The '\0' character is provided by the function. The textstring may be defined in an array, such as

```
char text[] "Just call me a text string.";
outtext(text);
```

or included in the function call

```
outtext("Gosh, I feel all strung out.");
```

The outtextxy() prototype

```
void far outtextxy(x,y,char far *textstring);
```

is similar to outtext();. Both functions display text in various fonts, sizes and colors. Table 7 of the previous article summarized the library functions for graphics mode text output. A description of these related functions is an article subject in itself and is not relevant to the topic at hand. What is relevant are stumbling blocks to be overcome in applying these two functions to our action programs.

The first is a lack of formatting. Recall that with printf() we can write a statement such as

```
printf("My name is %s.\n",name);
```

We can still display formatted strings but we must first use sprintf() in a manner of this sort:

```
sprintf(name_buf,"My name is %s.\n",name);
```

following which we can call

```
outtext(name_buf);
```

or

```
outtextxy(x,y,name_buf);
```

By the way, '\n' is supposed to work with sprintf(), and it does in the text mode but not in the graphics, from my own experience.

Well, making use of sprintf() doesn't appear too complex, and in fact it is not. But there is more to follow. In a game we mostly will want a banner somewhere displaying the player's score. Suppose it were to read

```
"Your score is now 1000"
```

and we do an update in steps of 100 with "score += 100;" somewhere in our program. We begin with

```
sprintf(score_buf,"You have %d points.",score);
```

with the assumption we can follow with

```
outtextxy(x,y,score_buf);
```

Surprise. The score updates all right. Only it quickly becomes unreadable because the update writes over the existing. I found no indication that this difficulty would arise in any of the references I have. It took several hours of angry exasperation to resolve the problem.

My initial attempt was to simply employ sprintf() to fill the buffer with a blank expression equal in length to the text in hopes that overwriting the text this way would wipe it out. Nope.

The ultimate solution was to enter the text in a viewport, display it, and, following each update, clear the viewport and rewrite it. The code has this form:

```
   setviewport(0,0,319,10,1); /* full screen width is not
necessary */
    sprintf((score_buf,"You have %d points.",score);
     outtextxy(0,0,score_buf);
```

This can be incorporated in a loop with selection logic for updating only as the situation requires. For example:

```
while(i) {  /* game maintenance loop */
 if(score_flg == 1) { /* set flag if score updated */
   clearviewport();   /* clear for score update */
setviewport(0,0,319,10,1);  /* restore viewport */
 sprintf((score_buf,"",score); /* update the score */
  outtextxy(0,0,score_buf);  /* display updated score */
    score_flg = 0;   }   /* reset the score flag */
     .
     .
     .
          }
```

If anyone has a better solution I would appreciate knowing of it. Several texts can be included in a single loop. There are five for the game I wrote. Each has its own sprintf() and outttextxy() definition. The viewport is full screen width by 20 pixels high.

Note that in this example I have set the clip value to 1. The program of Listing 6 has 0 instead. What I discovered, the hard way as is typical, is that unless the clip integer is zero outtext() will not function outside the viewport. Now that goes against some other things I have done, which leaves me scratching my head over reasons for. In the action game to be described in my next segment I display the scoring in a viewport very similar to this example and project images all over the screen. But not text.

(Continued on page 26)

Multitasking in Forth

With New Micros F68FC11 and Max-Forth

by Matthew Mercaldo

Some time ago I was working on a project for a small engineering firm. A friend of years past and I were developing a networked data acquisition system in Forth. This was the first time I had ever heard of Forth. Previous to this project the worlds of PASCAL, C and traditional assembler were the only I knew. Forth was so innovative, flexible and what my friend called extensible; truly a coherent medium to solve problems. Surely the only way to code! "The world would soon realize this wouldn't it?" My question would encourage my war torn friend. We'd fought for Forth on this project. The combination of my friend's wisdom in such matters, and my enthusiasm for this wondrous new way to solve problems won upper management to our camp. The speed at which we could develop sound code astonished me, and caused our immediate management embarrassment. Their alphabet never got past C and the innovation beyond. It was clear to me that Forth gave me the whole alphabet. My friend would always do alchemy with defining words, he proved to me that the simplest solution was the best. (Our product's performance demonstrated this.) There was always something new to learn. I would understand the implementation of concept in Forth, then I would grasp how a new piece of my friend's code worked. One concept that I understood, yet always amazed me was multitasking in Forth.

I recently decided to work through multitasking in Forth utilizing Max-Forth on the F68HC11 by New Micros Inc. The following article will detail the inner workings of two multitasking kernels along with some required support routines. I hope you enjoy experimenting with this paradigm as much as I have.

First I'll give you some conceptual jargon. Forth systems are inherently reentrant. This means that words written in a Forth kernel utilize stacks and registers; memory which can be retained by that context. Forth caters to multitasking in that all context specific information is referenced via the "UP" or User Pointer. This UP pointer points to a block of information. This block or User Area contains pointers to objects such as the Terminal Input Buffer, pointers to vocabularies, as well as a Dictionary Pointer, vectors pointing to Key and Emit routines, the original reset states for data and return stacks, etc. (See Figure one for Max-Forth's User area structure.) We can have many of these blocks, each block associated with a different instance of program, and a mechanism to switch to the next block upon request. This is the basis of a multitasking system in Forth. In order to facilitate this, Forth implementations will allow space in each user area for pointers to the next task user area and previous task user area. Max-Forth on the F68HC11 does this. To make a multi-

[Text continued on page 15]

Figure 1: Max-Forth's User area structure.

```
Max-Forth User Area Structure Members Required for Multitasking
Address (Hex)              Description

00:01                      Word Pointer
02:03                      Intruction Pointer
04:05                      User Pointer
06:07                      Next Task Pointer
08:09                      Previous Task Pointer
0A:0B                      Task Priority
0C:0D                      RPSAVE (this task's Stack Pointer)
0E:0F                      R0
10:11                      S0

16:17                      User Key Vector (Assembler Routine)
18:19                      User Emit Vector (Assembler Routine)
1A:1B                      User ?Terminal Vector (Assembler Routine)
1C:1D                      Terminal Input  Buffer Pointer (TIB)
1E:1F                      Terminal Input Buffer Size (C/L)

22:23                      User PAD

2C:2D                      Dictionary Pointer (DP)

34:35                      VOC-LINK
38:39                      UFORTH Link
40:41                      UEDITOR Link
46:47                      UASSEMBLER Link

4C:4D                      UABORT
54:55                      WARNING

6A:6B                      CONTEXT
6C:6D                      CURRENT
```

Figure 2: Memory allocation routines and hardware definitions.

```
HEX

050 100 TIB 2!
TIB 2@ + 010 + 022 !
FORGET TASK          ( Prepare to move dp to external rom. )


4000                 ( The size of the ROM. )
4000                 ( The starting address of ROM. )

DUP 4 + DP !         ( Put dictionary in external rom. )
                     ( 4 bytes reserved at start of rom for )
                     ( the autostart pattern followed by)
                     ( the cfa of the startup word. )
                     ( Now we can start defining new words. )

DUP CONSTANT ROM-Start
+   CONSTANT ROM-Limit

HERE ROM-Limit HERE - ERASE  ( Erase rom; unused space )
                             ( at end is 0Us  )

DECIMAL
20000 CONSTANT /10MS  ( 2000 ticks of free-running counter )
HEX
' /10MS @ 026 !      ( Point address 26 at eclock ticks per )
                     ( 10 ms. Contents of 26 was supposed to )
                     ( time writes to EEPROM but version 3.3 )
                     ( F68HC11 has bug--it uses 26 as a ptr. )

( _____RAM_____ )

( The first 256 bytes of RAM is on the 68HC11 chip and can be )
```

[Continued]

[Figure 2 continued]

```
( accessed more quickly than external RAM.                    )
( Variables are assigned to internal ram by $BYTE or $BYTES or )
( to external ram by BYTE or BYTES. For example,              )
(          $BYTE LOCK                                         )
( allocates a byte of internal ram for the RLOCKS and         )
(          12 BYTES BUFFER                                    )
( allocates 12 bytes of external ram to data named BUFFER     )

0073 CONSTANT $RAM-Start     ( first user                     )
0098 CONSTANT $RAM-Limit     ( $RAM-Limit-[S0]=030h bytes     )
VARIABLE '$RAM               ( ptr to next free byte of low ram )
    $RAM-Start '$RAM !

: RESET'$RAM   $RAM-Start '$RAM ! ;

: allot-$ram   ( n -- ; allot n bytes of direct ram )
    '$RAM +!  $RAM-Limit '$RAM @ U< ABORT"  no more internal ram" ;

: $BYTES  ( bytes <name> -- )    '$RAM @ CONSTANT   allot-$ram ;
: $BYTE   ( <name> -- )     1 $BYTES ;


0200 CONSTANT RAM-Start      ( Where to start ram allocation )
0FFF CONSTANT RAM-Limit      ( Ram allocation limit )

VARIABLE 'RAM ( pointer to next available external ram location )
    RAM-Start 'RAM !

: RESET'RAM ( -- )
    RAM-Start 'RAM !  ;

: allot-ram ( n -- ; allot n bytes of external ram )
    'RAM  +!  RAM-Limit  'RAM  @ U< ABORT" no more external ram" ;

: BYTES ( bytes <name> -- )   'RAM @ CONSTANT allot-ram ;
: BYTE ( <name> -- )   1 BYTES ;
```

```
: EE!   ( n addr -- )
   2DUP @ =
   IF  2DROP   ( was already equal to n )
   ELSE
       OVER 100 / OVER EEC!   1+ EEC!
   THEN ;

( _____Setting an Interrupt Vector_____ )

( Define each interrupt handler using CREATE, so that the name  )
( of the interrupt handler leaves the address of the code.      )
( Install the handler into the interrupt vector table by        )
(                    <code address> <vector name> VECTOR        )
( For example, to create pulse interrupt handler:               )
(                    CREATE PULSAR                              )
(                           ASSEMBLER                           )
(                           <code>                              )
(                           RTI                                 )

( Assume that the interrupt vector for PULSE is at address FEEE )
( in the  68HC11.  To get the word >PULSE:                      )
(                    FFFE @ CONSTANT >PULSE                     )
( Max-Forth points the interrupt vectors into the 11's EEPROM   )

( To install the PULSE interrupt handler:                       )
(                    : INSTALL-PULSAR                           )
(                           PULSAR >PULSE VECTOR ;              )
( The system reset code should then execute INSTALL-PULSAR.     )


: VECTOR   ( code-addr secondary-vector -- ;    )
       7E OVER EEC!    ( put jump opcode        )
       1+ EE! ;        ( put address            )
```

Figure 3:Code for timed interval tasker

```
( Multitasking module for 68HC11 : Interval Preemtive context switch )

( Matthew Mercaldo             )
( 41 South Park Street         )
( Oconomowoc, Wisconsin  53066 )

( MEM  1/27/90 )

HEX
( MaxForth world constants )
00 CONSTANT W                  ( Word Pointer )
02 CONSTANT IP                 ( Instruction Pointer )
04 CONSTANT UP                 ( User Area Pointer )

( Task area constants for control type task )
( I subtract 6 because of offset from start of this User Area )
06 6 -   CONSTANT NEXT_TASK    ( Pointer to the next task )
08 6 -   CONSTANT PREV_TASK    ( Pointer to the previous task )
0A 6 -   CONSTANT  PRIORITY    ( Task priority; not implemented )
0C 6 -    CONSTANT RPSAVE      ( This context's SP )
0E 6 -    CONSTANT R0          ( Return stack base )
10 6 -    CONSTANT DATA_STACK  ( Data stack base )

( used to hold the SP temporarily in setting task ready to run )
76 6 - CONSTANT STACK_HOLD

( vocabulary and link constants )
34 6 -    CONSTANT  UVOC-LINK
40 6 -    CONSTANT  UEDIT-LINK
46 6 -    CONSTANT  UASSEM-LINK
6A 6 -   CONSTANT  UCONTEXT
6C 6 -    CONSTANT  UCURRENT

( some 68HC11 hardware definitions )
B024 CONSTANT TMSK2
B025 CONSTANT TFLG2
80 CONSTANT TOI
80 CONSTANT TOF


CODE-SUB EI     ( enable interrupts )
       ASSEMBLER   CLI   RTS   END-CODE

CODE-SUB DI     ( disable interrupts )
       ASSEMBLER   SEI   RTS   END-CODE

( Set terminal task to point to itself. )
( This allows tasks to be linked into the round robin )
UP @ DUP   NEXT_TASK  + !
UP @ DUP   PREV_TASK + !

( Given the CFA of a forth word in X; NEXT2 will execute that word. )
( Does what MaxForth does, except now I know where it lives. )
CREATE NEXT2    ( -- ; X has CFA of word to run )
ASSEMBLER
       W ^ STX
       0 ,X LDX
       0 ,X JMP

( Sets the new task's processor stack. )
( This contains the task's context ready for a hit by pause. )
CODE SET_UP_CONTEXT  ( cfa task_blk -- )
ASSEMBLER
       0  ,Y LDX              ( task block base into register X )
       STACK_HOLD ,X  STS     ( remember what the SP is )
       R0 ,X LDS              ( get where rtn stack )

       NEXT2 # LDX  PSHX      ( PC:  next2 is first on stack )

       0 ,Y LDX   DATA_STACK ,X LDX ( get data stack and stack as Y )
       PSHX

       2 ,Y LDX          PSHX  ( get cfa and stack as X )

       0 # LDX                 ( stack a and b )
       PSHX

       40 # A LDA   A PSH      ( stack cc; inhibit xirq  )

       PSHX                    ( stack IP and W )
       PSHX

       0 ,Y LDX  RPSAVE ,X STS ( set rp save pointing to where SP is )

       STACK_HOLD ,X  LDS      ( restore original stack )

       INY INY INY INY         ( clean up forth data stack )
       NEXT ^ JMP
END-CODE

( Connects the vocabulary pointers for this task. )
( These are not neccessary for a control task )
( but are added if another terminal task is wanted. )
( To make a terminal task... connect in new : )
( UPAD, UTIB, UC/L. )
( To be careful, revector key and emit. )
CODE  SET_VOC ( task_blk -- task_blk )
ASSEMBLER
       0 ,Y LDX
       UVOC-LINK ,X LDD        0 ,Y ADDD   UVOC-LINK ,X STD
       UEDIT-LINK ,X LDD       0 ,Y ADDD   UEDIT-LINK ,X STD
       UASSEM-LINK ,X LDD   0 ,Y ADDD   UASSEM-LINK ,X STD
       UCONTEXT ,X LDD            0 ,Y ADDD   UCONTEXT ,X STD
       UCURRENT ,X LDD          0 ,Y ADDD   UCURRENT ,X STD
       NEXT ^ JMP
END-CODE

: SET_R0 ( task_blk -- task_blk )
       DUP  R0 + 'RAM @  SWAP !   ;
: SET_S0  ( task_blk -- task_blk )
```

[Continued]

[Figure 3 continued]

```
        DUP  DATA_STACK +  'RAM @  SWAP !   ;

CREATE (LINK_TASK)  ( task_blk -- task_blk )
( points this task_blk to the terminal task_blk )
ASSEMBLER
        UP ^ LDX     PREV_TASK ,X  LDD
        0 ,Y LDX      PREV_TASK ,X  STD
        UP ^ LDD     NEXT_TASK ,X   STD
        RTS

CODE LINK_TASK ( task_bock -- task_block ; connects )
                ( this User Area into list )
ASSEMBLER
        (LINK_TASK) ^ JSR
        NEXT ^ JMP
END-CODE

:  ::  ( cfa stack  <name> -- ; create a task )
( On command line to create a task )
(  ' <word> CFA  <stack_size>  ::  <task_name>  ;;  )
( :: is the task definition word. )
( It allots the proper number of bytes for a task_blk. )
( It allots 100 hex bytes for return stack. )
( It allots the desired bytes for data stack. )
        'RAM @  DUP      200 BYTES
        UP @  SWAP  100 6 -  CMOVE
        SET_R0
        SWAP allot-ram
        SET_S0
        LINK_TASK
        SET_VOC
        SET_UP_CONTEXT  ;
( to set task running, do WAKE command )

: ;; ;  ( -- ; added for continuity of syntax )

( Pause switches context. )
( The registers are stacked by interrupt. )
( The Forth IP and W pointers are stacked. )
( The SP is saved in this tasks context. )
( The next task SP is installed. )
( IP and W for next task are taken from the next task's )
( stack.  Next task is resumed by RTI )
CREATE  ~PAUSE  ( -- ;context switch )
ASSEMBLER
        ( acknowledge the timer overflow interrupt )
        TOF # A LDA   TFLG2 ^ A STA

        ( stack IP and W for this task )
        IP ^ LDX   PSHX
        W ^ LDX   PSHX

        ( set this task's Stack pointer into RPSAVE )
        UP ^ LDX
        RPSAVE ,X STS

        ( Get next task )
        NEXT_TASK ,X  LDX
        UP ^ STX

        ( restore next task's Stack pointer )
        RPSAVE ,X LDS
        ( restore next task's W and IP )
        PULX   W ^ STX
        PULX  IP ^ STX

        ( resume execution of next task )
        RTI


( Wake connects the task_blk to the round robin )
CREATE (WAKE)  ( task_blk -- )
ASSEMBLER
        0 ,Y LDX    NEXT_TASK ,X LDX    PREV_TASK ,X LDX
        0 ,Y LDD    NEXT_TASK ,X STD

        0 ,Y LDX    NEXT_TASK ,X LDX    PREV_TASK ,X LDD
        0 ,Y LDX   PREV_TASK ,X STD

        NEXT_TASK ,X LDX    0 ,Y LDD   PREV_TASK ,X STD

        INY INY
        RTS

( Sleep disconnects the task_blk from the round robin. )
( The task_blk points to it's previously connected tasks, )
( but they no longer point to it. )
CREATE (SLEEP)  ( task_blk -- )
ASSEMBLER
        0 ,Y LDX   NEXT_TASK ,X LDD   PREV_TASK ,X LDX
        NEXT_TASK ,X STD

        0 ,Y LDX   PREV_TASK ,X LDD   NEXT_TASK ,X LDX
        PREV_TASK ,X STD

        INY INY
        RTS

CREATE (sleep)  ( -- ;puts self to sleep on next pause )
ASSEMBLER
        UP ^ LDX  NEXT_TASK ,X LDD   PREV_TASK ,X LDX
        NEXT_TASK ,X STD

        UP ^ LDX   PREV_TASK ,X LDD   NEXT_TASK ,X LDX
        PREV_TASK ,X STD
        RTS

CODE sleep ( -- ;puts self to sleep)
ASSEMBLER
        (sleep) ^ JSR
        NEXT ^ JMP
END-CODE

CODE WAKE ( task_blk -- ; wake this task )
ASSEMBLER
        (WAKE) ^ JSR
        NEXT ^ JMP
END-CODE

CODE SLEEP ( task_blk -- ; sleep this task)
ASSEMBLER
        (SLEEP) ^ JSR
        NEXT ^ JMP
END-CODE

( Stops the tasking interrupt )
: TASKER_STOP  ( -- )
        TMSK2 C@   TOI  1 - AND     TMSK2 C! ;
( Starts the tasking interrupt )
: TASKER_GO  ( -- )
        TMSK2 C@   TOI OR     TMSK2 C!  ;

( High level forth task wake and task sleep )
: WAKE  ( task_blk -- )
        TASKER_STOP  WAKE  TASKER_GO  ;

: SLEEP  ( task_blk -- )
        TASKER_STOP  SLEEP TASKER_GO   ;

( ffde points to eeprom inside 68hc11 )
( there a jmp can be added to the interrupt )
FFDE @  CONSTANT >TOI

( Tasking_on sets interrupt vector and )
        ( turns on PAUSE interrupt )
: TASKING_ON
        DI
        ~PAUSE  >TOI  VECTOR
        TOI TMSK2 C!
        EI   ;


( Test cases )

VARIABLE TIMER1
0 TIMER1 !
VARIABLE TIMER2
0 TIMER2 !

: TEST1  EI   1 TIMER1 +!
        BEGIN 1 TIMER1 +!   AGAIN ;

: TEST2   EI   1 TIMER2 +!
        BEGIN 1 TIMER2 +!  AGAIN ;


( create task one with 20 data stack bytes running TEST1 )
' TEST1 CFA 20 :: TASK1 ;;
( start task one running )
TASK1 WAKE

( create task two with 20 data stack bytes running TEST2 )
' TEST2 CFA 20 :: TASK2 ;;
( start task two running )
TASK2 WAKE

( Do  a TIMER1 @ .  a few times to see task one running )
( Do a TIMER2 @ . a few times to see task two running )
```

Figure 4:Code for the cooperative tasker.

```
( Multitasking module for 68HC11 : Cooperative Tasker )

( Matthew Mercaldo              )
( 41 South Park Street          )
( Oconomowoc, Wisconsin  53066 )

( MZM  2/3/90 )

HEX
( MaxForth world constants )
00 CONSTANT W                   ( Word Pointer )
02 CONSTANT IP                  ( Instruction Pointer )
04 CONSTANT UP                  ( User Area Pointer )

( Task area constants for control type task )
06 6 -   CONSTANT NEXT_TASK     ( Pointer to the next task )
08 6 -   CONSTANT PREV_TASK     ( Pointer to the previous task )
0A 6 -   CONSTANT  PRIORITY     ( Task priority; not implemented )
0C 6 -    CONSTANT RPSAVE       ( This context's SP )
0E 6 -    CONSTANT R0           ( Return stack base )
10 6 -    CONSTANT DATA_STACK   ( Data stack base )

( used to hold the SP temporarily in setting task ready to run )
76 6 - CONSTANT STACK_HOLD

( vocabulary and link constants )
34 6 -    CONSTANT  UVOC-LINK
40 6 -    CONSTANT  UEDIT-LINK
46 6 -    CONSTANT  UASSEM-LINK
6A 6 -   CONSTANT  UCONTEXT
6C 6 -    CONSTANT  UCURRENT

( set terminal task to point to itself )
( this allows tasks to be linked into the round robin )
UP @ DUP   NEXT_TASK  + !
UP @ DUP   PREV_TASK + !

( Given the CFA of a forth word in X; NEXT2 will execute that word. )
( Does what MaxForth does, except now i know where it lives. )
CREATE NEXT2    ( -- ; X has CFA of word to run )
ASSEMBLER
        W ^ STX
        0 ,X LDX
        0 ,X JMP

( Sets the new task's processor stack. )
( This contains the task's context ready for a hit by pause. )
CODE SET_UP_CONTEXT  ( cfa task_blk -- )
ASSEMBLER
        0  ,Y LDX               ( task block base into register X )
        STACK_HOLD ,X  STS      ( remember what the SP is )
        R0 ,X LDS               ( get where rtn stack )

        NEXT2 # LDX  PSHX       ( PC:  next2 is first on stack )

        0 ,Y LDX  DATA_STACK ,X LDX ( get data stack and stack as Y )
        PSHX

        2 ,Y LDX          PSHX  ( get cfa and stack as X )

        0 # LDX                 ( stack a and b )
        PSHX

        40 # A LDA   A PSH      ( stack cc; inhibit xirq  )

        PSHX                    ( stack IP and W )
        PSHX

       0 ,Y LDX RPSAVE ,X STS ( set rp save pointing to where SP is )

        STACK_HOLD ,X  LDS      ( restore original stack )

        INY INY INY INY         ( clean up forth data stack )
        NEXT ^ JMP
END-CODE

( Connects the vocabulary pointers for this task )
( these are not neccessary for a control task )
( but are added if another terminal task is wanted. )
( To make a terminal task... connect in new : )
( UPAD, UTIB, UC/L. )
( To be careful, revector key and emit. )
CODE  SET_VOC ( task_blk -- task_blk )
ASSEMBLER
        0 ,Y LDX
        UVOC-LINK ,X LDD        0 ,Y ADDD   UVOC-LINK ,X STD
        UEDIT-LINK ,X LDD       0 ,Y ADDD   UEDIT-LINK ,X STD
        UASSEM-LINK ,X LDD   0 ,Y ADDD   UASSEM-LINK ,X STD
        UCONTEXT ,X LDD           0 ,Y ADDD   UCONTEXT ,X STD
        UCURRENT ,X LDD          0 ,Y ADDD   UCURRENT ,X STD
        NEXT ^ JMP
END-CODE

: SET_R0 ( task_blk -- task_blk )
        DUP  R0 + 'RAM @  SWAP !   ;
: SET_S0  ( task_blk -- task_blk )
        DUP  DATA_STACK +  'RAM @  SWAP !   ;

CREATE (LINK_TASK)  ( task_blk -- task_blk )
( points this task_blk to the terminal task_blk )
ASSEMBLER
        UP ^ LDX    PREV_TASK ,X  LDD
        0 ,Y LDX     PREV_TASK ,X  STD
        UP ^ LDD    NEXT_TASK ,X   STD
        RTS

CODE LINK_TASK ( task_bock -- task_block )
ASSEMBLER
        (LINK_TASK) ^ JSR
        NEXT ^ JMP
END-CODE

:  ::  ( cfa stack  <name> -- ; create a task )
( On command line to create a task )
(  ' <word> CFA  <stack_size>  ::  <task_name>  ;;  )
( :: is the task definition word. )
( It allots the proper number of bytes for a task_blk. )
( It allots 100 hex bytes for return stack. )
( It allots the desired bytes for data stack. )
        'RAM @  DUP     200 BYTES
        UP @  SWAP  100 6 -  CMOVE
        SET_R0
        SWAP allot-ram
        SET_S0
        LINK_TASK
        SET_VOC
        SET_UP_CONTEXT  ;
( to set task running, do WAKE command )

: ;; ;  ( -- ; for syntactical continuity )

( Pause switches context. )
( The registers are stacked by interrupt. )
( The Forth IP and W pointers are stacked. )
( The SP is saved in this tasks context. )
( The next task SP is installed. )
( IP and W for next task are taken from the next task's )
( stack.  Next task is resumed by RTI )
CREATE  >PAUSE>  ( -- ;context switch )
ASSEMBLER
        ( JSR has put PC on the stack )
        PSHY                    ( Y on next )
        PSHX                    ( X on next )
        A PSH                   ( A on next )
        B PSH                   ( B on next )
        TPA  A PSH              ( CC on next )

        IP ^ LDX   PSHX         ( IP on next )
        W ^ LDX   PSHX          ( W on next )

        UP ^ LDX                ( remember SP for this task area )
        RPSAVE ,X STS           ( in RPSAVE )

        NEXT_TASK ,X  LDX       ( Get pointer to next task area )
        UP ^ STX        ( set the UP pointer pointing to next task )

        RPSAVE ,X LDS  ( get next task's SP from next task's RPSAVE )

        PULX   W ^ STX          ( restore W )
        PULX  IP ^ STX          ( restore IP )

        RTI                     ( restore registers )

CODE PAUSE ( -- ; switch context to next task in round robin )
ASSEMBLER
        >PAUSE> ^ JSR           ( switch context )
        NEXT ^ JMP              ( The above JSR puts PC here. )
                              ( This PC is that of the new context. )
END-CODE


( Wake connects the task_blk to the round robin )
CREATE (WAKE)  ( task_blk -- )
ASSEMBLER
        0 ,Y LDX    NEXT_TASK ,X LDX    PREV_TASK ,X LDX
        0 ,Y LDD    NEXT_TASK ,X STD

        0 ,Y LDX    NEXT_TASK ,X LDX    PREV_TASK ,X LDD
        0 ,Y LDX   PREV_TASK ,X STD

        NEXT_TASK ,X LDX    0 ,Y LDD   PREV_TASK ,X STD

        INY INY
        RTS

( Sleep disconnects the task_blk from the round robin. )
( The task_blk points to it's previously connected tasks, )
( but they no longer point to it. )
CREATE (SLEEP)  ( task_blk -- )
ASSEMBLER
```

[Continued]

[Figure 4 continued]

```
        0 ,Y LDX   NEXT_TASK ,X LDD   PREV_TASK ,X LDX
        NEXT_TASK ,X STD

        0 ,Y LDX   PREV_TASK ,X LDD   NEXT_TASK ,X LDX
        PREV_TASK ,X STD

        INY INY
        RTS

CREATE (sleep)  ( -- ;puts self to sleep on next pause )
ASSEMBLER
        UP ^ LDX  NEXT_TASK ,X LDD   PREV_TASK ,X LDX
        NEXT_TASK ,X STD

        UP ^ LDX   PREV_TASK ,X LDD   NEXT_TASK ,X LDX
        PREV_TASK ,X STD
        RTS

CODE sleep ( -- ;puts self to sleep)
ASSEMBLER
        (sleep) ^ JSR
        >PAUSE> ^ JSR
        NEXT ^ JMP
END-CODE

CODE WAKE ( task_blk -- )
ASSEMBLER
        (WAKE) ^ JSR
        NEXT ^ JMP
END-CODE

CODE SLEEP ( task_blk -- )
ASSEMBLER
        (SLEEP) ^ JSR
        NEXT ^ JMP
END-CODE

( Test cases )
VARIABLE TIMER1
0 TIMER1 !
CREATE TIMER2  0 C,
0 TIMER2  C!

: TEST1 ( -- )
        BEGIN PAUSE 1 TIMER1 +!  AGIAN ;

CODE TEST2 ( -- )
ASSEMBLER
        TIMER2 ^ INC
        BEGIN
                >PAUSE> ^ JSR
                TIMER2 ^ INC
        AGAIN
END-CODE

( Create Task1 with 20 data stack bytes running TEST1 )
' TEST1 CFA  20 :: TASK1 ;;
( Wake task1 )
TASK1 WAKE

( Create Task2 with 20 data stack bytes running TEST2 )
' TEST2 CFA 20 :: TASK2 ;;
( Wake task2 )
TASK2 WAKE

( To see this tasking, type PAUSE, then watch the timers increment )
( PAUSE TIMER1 @ . TIMER2 C@ .  )
```

User Disk

The code from *Multitasking in Forth, Animation with Turbo C, Mysteries of PC Floppy Disks*, and *Real Computing*, is available on a 5.25" 360K or 3.5" 720K PC format disk for $10 postpaid in the U.S.

tasker we need a mechanism to first define a task and allot the required memory, wake a task, put a task to sleep, and we need the word to switch context to the next task in the list.

There are two ways that I will demonstrate multitasking. The first and simplest utilizes a regular timed interrupt to switch context. The second tasking technique we'll call "Cooperative Tasking". In cooperative tasking a word is defined which when executed gives the CPU to the next task in the list of tasks. Cooperative tasking is the most demanding on the system engineers talents. The programmer carefully crafts the context switching word (usually called PAUSE) into the words of the system. The multitasking occurs at a non-interrupt level.

I prefer Cooperative tasking because by it the programmer / architect can craft the priority of tasks into his/her system. I also see the non-preemptive nature of cooperative tasking as beneficial because we don't waste valuable interrupt resources on tasking.

In Figure two we see some routines necessary for memory allocation and some hardware definitions. The assembly code can be compiled in by hand. (The reader may also purchase the assembler used in this article from the author.) Figure three is the actual code for the timed interval tasker. Figure four is the actual code for the cooperative tasker.

In order to download the multitasker kernel, we must first build some "tool" words. These words are described in the multitasking startup module (Figure two). The multitasking module contains the initial Max-Forth environment setup, memory allotment words, and interrupt revectoring words. The environment used assumes external RAM from 0100 hex to 7FFF hex. I use a 32Kx8 static ram in socket U2 of the New-Micros NMIT/X series boards.

The very first environment initialization performed is moving the Terminal Input Buffer (TIB) to external RAM and setting the TIB size to 80 characters. Next the dictionary pointer (DP) is set to point to address 4000 hex. This can be altered for your hardware configuration. Next the EEPROM write timing bug is fixed. There was an indirection problem associated with the variable used to time writing of EEPROM in Max-Forth V3.3. Space for internal and external ram allotment can be adjusted by setting $RAM-Start, $RAM-Limit, RAM-Start and RAM-Limit to accommodate your hardware configuration. VECTOR is a primitive version of a word that I use. This version will not write EEPROM properly while interrupts are enabled.

The comments in the code explain the code. Have fun, and happy Forthing!

You can purchase the assembler from me for $50 at the address given in this article. Please specify PC or MacIntosh. •

Max-Forth is a trademark of New Micros Inc.

MacIntosh is a trademark of Apple Computers.

Matthew Mercaldo
41 South Park Street
Oconomowoc, WI 53066

Editor's Note: The code listings were prepared on a MAC and when I placed them with the '286, PageMaker regurgitated type all over the page. I reformatted the listings, and had to squeeze the width to fit on the page. Any formatting errors are my responsibility, and not the author's

---

Mysteries of PC Floppy Disks Revealed

by Richard Rodman

Finally, a nationwide magazine dares to defy the conspiracy and reveals the shocking story behind PC floppy disks! Don't rush ahead to the earth-shattering conclusion—all of your questions will be answered in turn. First, some history and basic theory.

FM, MFM and GCR

Over ten years ago, there were no single-chip floppy disk controllers. Floppy disk controllers were big, complex circuit boards. Most systems used hard sectoring—the sectors were separated by holes punched in the disk.

However, IBM realized that soft sectoring—separating sectors by software-generated data patterns on the disk—was a more powerful technique. In the late seventies, Western Digital developed the first single-chip floppy disk controller, the WD-1771. This device is probably just as responsible for the personal computer boom as the Intel 8080.

The FD-1771 wrote data in what was called FM, "Frequency Modulation." This is a recording technique in which each data bit is recorded with a clock pulse, followed by a data pulse or the absence thereof. Today, this technique is known as "single density."

FM was wasteful, because two bits of magnetic storage were being used to hold one bit of data. MFM, "Modified Frequency Modulation," was developed to correct this waste. By cleverly writing a pulse for each bit dependent on what the last bit was, they doubled the actual data rate. Today, this technique is known as "double density."

In the meantime, it appears that a garage computer designer named Wozniak either didn't know about the FD-1771 or couldn't afford one, so he put together a little circuit wherein a single clock bit sufficed for a number of data bits, and used the CPU to generate this pattern in software. This technique is called GCR, for "Group Coded Recording." It achieves a density somewhat higher than FM, but lower than MFM.

Because GCR differs at the bit level, it is fundamentally incompatible with both FM and MFM, and this incompatibility has become the tyrannical hobgoblin of Apple Computer. It survives to this day in the form of a chip called the IWM used in the Macintosh. This is why no Apple machine can read a disk from any other computer, nor can any other computer read a disk from an Apple.

An incompatible form of GCR is also used by Commodore computers.

The Boot Sector

In the old days of CP/M, there was no standard format for 5 inch floppies. A BIOS writer generally made an educated guess based on how many bytes were in a sector.

Many mechanical systems have been proposed and implemented. For example, 8 inch floppies have an offset index hole if they were double-sided. More recently, some 3.5 inch drives look for a hole on the opposite side of the disk to see if the media is high-density-capable. (Actually, I think very few drives do that.)

But most PCs identify the format of a diskette by looking in the first sector of the disk. There, they find a data structure called the Boot Parameter Block (BPB). This data structure contains parameters which are used in the operating system, such as how many sectors there are per track, how many sectors per cluster, and so on.

Few people know it, but the Atari ST is really running a 68000 clone of MS-DOS. While it will read the 720K MS-DOS format without modification, MS-DOS will only read drives with BPBs that it recognizes (and not Atari's). Programs are available for the ST which will format disks in PC-compatible formats. (Only one parameter is incorrect—see Table 1).

Table 1 shows the BPB values of the most common MS-DOS formats. The SSDD 8 sector, DOS 1.0 format (/1/8) is not shown because a BPB is not written in that format. BPBs were not really added until DOS 3.0.

Listing 1 is a simple program which displays the BPB of any drive. Have fun!

There is also a partition table in the boot sector of hard disks, but it is not used on floppies.

The 1.2 Megabyte 5" Disk

The first "high density" drive was the IBM 1.2MB 5" floppy. This drive arose out of a requirement to make a five-inch floppy that would closely emulate an eight-

Table 1 - BPB's of Common Formats

	Formats						
	360K 5" IBM	1.2M 5" IBM	720K 3.5" IBM	1.44M 3.5" IBM	180K 5" PCjr	360K 3.5" At.ST	720K 3.5" At.ST
Bytes/sector	512	512	512	512	512	512	512
Sectors/cluster	2	1	2	1	1	2	2
Resvd. sectors	1	1	1	1	1	1	1
Copies of FAT	2	2	2	2	2	2	2
Root dir. entries	112	224	112	224	64	112	112
Sectors/disk	720	2400	1440	2880	360	720	1440
Format ID	FD	F9	F9	F0	FC	F8	F9
Sectors/FAT	2	7	3	9	2	5	5
Sectors/track	9	15	9	18	9	9	9
Sides	2	2	2	2	1	1	2
Special res. sec.	0	0	0	0	0	0	0

Note that the format ID in the BPB may not necessarily agree with the format ID found in the first byte of the first FAT.

inch floppy. You see, five-inch floppies rotate at 300 RPM and write data in MFM at 250K bits/second, whereas eight-inch floppies rotate at 360 RPM and write data in MFM at 500K bits/second. Another difference was that five-inch drives have 40 tracks, whereas eight-inch drives have 77 tracks.

Increasing the number of tracks was simple, requiring only a more precise stepper motor. In fact, 80 track (96 tracks per inch) five-inch floppy drives were available before the IBM PC came out, and were a popular upgrade for the TRS-80. Before the IBM PC, the most popular five-inch drives were 40-track single-sided, single or double density, and 80-track double-sided double-density, which was often called "quad density."

While the first IBM PCs came out with the common 40-track single-sided double-density drive, storing 160K bytes, they soon standardized on a 40-track double-sided double-density drive storing 360K bytes, writing 9 sectors of 512 bytes on each track.

Incidentally, these 8 or 9 sectors were scoffed at by other computer system manufacturers as lazy and wasteful. The Kaypro II, for example, stored 10 sectors of 512 bytes on each track; the Osborne stored 5 sectors of 1024 bytes on each track.

The drive makers continued working on their project, however, which was presumably geared toward converting equipment manufacturers who perversely continued to use eight-inch drives. Eight-inch drives were still faster —they had a higher data rate and faster track stepping. Still, the drive manufacturers failed to anticipate how rapidly the eight-inch floppy market would collapse. By the time they had succeeded in making the eight-inch floppy drive emulator, there was little point in selling it as such. IBM picked it up, however, and it became the 1.2 MB floppy for the PC-AT.

So, the secret of the 1.2 MB drive is simply this: It is an 80-track, double-sided drive, that, when pin 2 is pulled low, increases its rotational speed to 360 RPM. At the same time, the NEC 765 floppy controller doubles its data rate, to 500K bits/second, and writes 15 sectors of 512 bytes per sector.

In some systems with 1.2MB drives, rather than both change the speed and double the data rate, the data rate is changed to an intermediate value of 300K bits/sec.

```
Listing 1 - Program to read BPB

/* Read drive BPB.  Datalight Optimum C.

     IMPORTANT! COMPILE WITH -a TO DISABLE STRUCTURE ALIGNMENT GARBAGE!
*/

#include ''dos.h''
#include ''stdio.h''

#define BYTE unsigned char
#define WORD unsigned int

struct bpb {
     BYTE     jump[ 3 ];
     BYTE     system_id[ 8 ];          /* e.g. IBM 2.1 */
     WORD     bytes_per_sector;
     BYTE     sectors_per_cluster;
     WORD     reserved_sectors;
     BYTE     copies_of_fat;
     WORD     root_dir_entries;
     WORD     sectors_on_disk;     /* clusters? */
     BYTE     format_id;
     WORD     sectors_per_fat;
     WORD     sectors_per_track;
     WORD     sides;
     WORD     special_reserved_sectors;
};

BYTE buffer[ 512 ];

main() {
     int i, n;
     struct bpb *p;

     for(;;) {
          puts( ''enter drive (0=A, 1=B, 2=C, etc.)(-1 to quit):'' );
          scanf( ''%d'', &n );
          if( n < 0 ) break;

          n = dos_abs_disk_read( n, 1, 0, &buffer[ 0 ] );

          printf( ''result of read was %d\n'', n );

          /* cumbersome - assign p to address of buffer base */

          p = ( struct bpb * ) &buffer[ 0 ];

          /* C is cumbersome here too... */

          printf( ''system id: %c%c%c%c%c%c%c%c\n'',
               p -> system_id[ 0 ],     p -> system_id[ 1 ],
               p -> system_id[ 2 ],     p -> system_id[ 3 ],
               p -> system_id[ 4 ],     p -> system_id[ 5 ],
               p -> system_id[ 6 ],     p -> system_id[ 7 ] );

          printf( ''   bytes per sector: %u\n'', p -> bytes_per_sector );
          printf( ''sectors per cluster: %u\n'', p -> sectors_per_cluster );
          printf( ''   reserved sectors: %u\n'', p -> reserved_sectors );
          printf( ''      copies of fat: %u\n'', p -> copies_of_fat );
          printf( ''   root dir entries: %u\n'', p -> root_dir_entries );
          printf( ''    sectors on disk: %u\n'', p -> sectors_on_disk );
          printf( ''          format id: %02x\n'', p -> format_id );
          printf( ''    sectors per fat: %u\n'', p -> sectors_per_fat );
          printf( ''  sectors per track: %u\n'', p -> sectors_per_track );
          printf( ''              sides: %u\n'', p -> sides );
          printf( '' special resvd secs: %u\n'', p -> special_reserved_sectors );
     }
}

/* end of readbpb.c */
```

The 1.44 Megabyte 3.5" Disk

The 3.5 inch drives are pretty much small copies of the 5 inch drives. This is the result of a standardization effort in the early 80s that resulted in an ANSI standard for 3.5 inch media and drives.

When IBM decided to change to the 3.5 inch drives, they realized they couldn't just give people the same 360K and 1.2MB sizes. Instead, they settled on two

compatible drives—an 80-track double-sided double-density mode (the "quad density" mode of yore), and a similar drive with a "high density" mode. The lower-density format uses nine 512-byte sectors per track, giving 720K bytes on a disk. This format is exactly the same as that used on 5 inch drives by some other manufacturers, such as Sanyo.

The 3.5 inch drives don't have an eight-inch drive emulation mode, however, so, unlike the 1.2MB drive, the 1.44MB drive doesn't change speed. The data rate is doubled as before, so that the 2MHz (500K bits/sec) data rate is used. Eighteen 512-byte sectors are used per track, instead of nine.

Having this information, you can appreciate why you shouldn't use normal-density media in high-density drives. High-density media isn't just normal-density media that tested out better. The 1.2MB media has to be able to reliably handle 20 percent more flux changes per inch—and the 1.44MB media has to reliably handle 100 percent more. While low-density media might seem to "work okay", the bigger magnetic particles on the normal-density material will have both weak magnetization and strange data pattern sensitivities.

Speaking of data pattern sensitivities, the worst-case value for MFM data storage is 6DB hex, because this pattern writes the fewest pulses on the disk and requires the most careful clock recreation. Whenever you format your disks, if you have a choice, use 6DB or a variation thereof, such as the two-byte value, 6D B6. Never be content with simply writing E5.

The relevant parameter seems to be something called "coercivity." Regular 360K floppies have a coercivity of 320 oersteds, and 1.2MB floppies have 640. Of course, you already know not to try to put 1.2MB on a low-density floppy. On the 3.5s, the 720K disks are 600 oersteds, and the 1.44MB is 640. I was surprised to see that they're so close—but, again, don't get the idea that the 720K disks will be "passable" at 1.44MB.

I've been dismayed to find that even name-brand 1.44MB diskettes have some bad sectors. This tends to make DISKCOPY operations iffy. However, some very inexpensive generics I purchased recently for under a dollar have formatted perfectly every time. Look for the ones that show you the oersteds right in the ad.

It should be possible to use ten 1024-byte sectors in high-density mode. This would give a drive capacity of 1.6 MB. Somewhere in the bowels of MS-DOS, however, there is a 512-byte sector size limitation, so don't bother trying it.

Table 2 - PC Floppy Interface Signals

Pin	Floppy Drive	PC Controller
2	High density (1.2M drives)	High density (if used)
4	not used	not used
6	DS4	not used
8	Index	Index
10	DS1	Motor on A:
12	DS2	DS2 (Drive select B:)
14	DS3	DS1 (Drive select A:)
16	Motor On	Motor on B:
18	Direction	Direction
20	Step	Step
22	Write data	Write data
24	Write gate	Write gate
26	Track 0	Track 0
28	Write protect	Write protect
30	Read data	Read data
32	Side 1 select	Side 1 select
34	Disk change (3.5" drives)	Disk change (if used)

Notes:

1. All odd pins are grounded.

2. The cable twists pins 10 to 16 between the connectors for the B (straight through) and A drive.

3. All floppy drives are jumpered for DS2.

4. Pin 2, on a 1.2M 5" floppy, changes rotational speed from 300 to 360 RPM. It has no effect on other drives. Some controllers don't use this technique to change density (see text).

5. Pin 34, on some 3.5" drives, goes low when no disk is present in the drive. To my knowledge, this signal has never been used by anyone. A more universal technique is looking for index pulses from the rotating diskette to see if it is present.

Why do PC Floppy Cables Have That Weird Twist?

Floppy drives were originally designed to be connected in parallel on a long, flat cable. The signals were driven by high-current, open-collector drivers. These drivers allowed for cable lengths of twenty feet or more; the last drive on the cable had a terminator resistor pack (all other drives had to have it removed). Four drive select signals were implemented to allow up to four drives on a cable, and each drive was jumpered to be selected by one of *DS1, DS2, DS3* or *DS4*. Five-inch drives also had a *Motor on* signal.

When IBM designed their PC, they realized that the conventional way of dealing with floppy drives made cable construction easy, but made it hard to configure multiple drives: The first drive had to have *DS1* set, but the second had to have *DS2*, and whichever was on the end of the cable had to have a terminator pack, and the other drive had to not have one. If anything was wrong, you've got trashed disks, lost data and very unhappy customers. A bad scene.

The first thing they wanted to do was to come up with a way to jumper all the drives the same. This they accomplished by the twist in the cable: pins 10 to 16 are twisted between the two drive connectors. Then, all drives are jumpered to *DS2* (drive two). The controller now has two drive select lines, one being *DS2*, pin 12, which selects drive B:, and one being *DS3*, pin 14, which selects drive A:. Also, there are now two *Motor on* lines instead of one, one being the original *Motor on*, pin 16, which now turns on drive B:'s motor, and the other one being the original *DS1*, pin 10. An unfortunate consequence of this kludge—which it is, to be sure—is that only two drives can be controlled instead of four. (See table 2).

That being done, after a couple of years, the realization dawned that floppy cables were short, only a couple of feet, and the high drive requirement was unnecessary. This allowed for lower-current (LS TTL) drivers, and higher resistance pull-ups which could be installed on both floppies. Today, there is no need to consider drive jumpering or terminating resistors at

(Continued on page 20)

DosDisk

The MS-DOS Disk Format Emulator

by Daniel J. Mareck

For the second time in two years, my good old work horse has been pulled from the brink. This time the discovery of DosDisk from Plu*Perfect has given my good ol' S100 workhorse a new lease on life. The last time it almost went to pasture was before I discovered ZCPR, but if I get carried away on that you'll never find out about DosDisk.

If you're faced with the dilemma of having all DOS-based PCs in your workplace, a CP/M system at home, and the need to make them communicate, then DosDisk is for you. It is a resident system extension (RSX) that allows you to access a standard 360k DOS-format floppy as if it were a CP/M disk native to your machine! This means using your favorite directory program, copy program, and even wordprocessor directly on the DOS-formatted file. The other 'exchange' programs I'm aware of require you to copy the file onto a native disk before operating on it.

The only one real restriction to DosDisk is that the disk must have previously been formatted in some other way. DosDisk cannot format a disk. As a side note, so I do not leave any doubt, DosDisk is NOT an emulator program. It will not allow you to 'run' DOS programs on your CP/M system.

Included with DosDisk is a utility that allows you to create and delete subdirectories and to select the directory to work with. This provides direct access to any subdirectory on the disk as well as the root directory. Several other utilities are provided that can take advantage of the date and time stamping of the DOS files. These include an enhanced directory program and a filesweep program. It should be noted, however, that for these programs to make effective use of the time stamps you must be running Plu*Perfect's DateStamper or one of the new ZDOSes.

The documentation that comes with DosDisk explains how to use it and each of the utilities provided with it. I found this documentation to be (for the most part) concise, clear, and complete. The 'kit' installation section was slightly confusing until I really started into the installation; then it became clear. Since that kind of thing seems to happen to me a lot, I don't think anything of it. My only real gripe is that the manual doesn't fit into an 8½ x 11 binder; it's made for an 'IBM' binder.

DosDisk is available preconfigured for several systems, including Kaypros, Morrows, Xeroxes, the SB180, and others. (Contact Z Systems Associates or Plu*Perfect Systems for details.) It is capable of running under CP/M 2.2, CP/M 3.0, and the Z-Systems. For those of us not fortunate enough to have one of the preconfigured target machines, Plu*Perfect also provides a 'kit' version. In the next section of this review, I will discuss in more detail what you have to do to make this 'kit' version run on your computer.

Dan Mareck is a computer systems engineer working as a consultant. He has over 15 years experience in real-time hardware/software systems design and is currently involved in software development for RISC processors embedded in radar ystems.

Dan is involved in many activities including the Lancaster Micro Computer User's Group (a FOG AMO) of which he is president. His spare time is spent tinkering with his home-brewed S100 system (running ZCPR of course!).

To get in touch with Dan electronically leave a message on GENIE for D.MARECK1, by voice at (717) 235 6568 and by Pony Express at 1 Westwoods Rd. New Freedom, PA 17349.

Building The DosDisk Kit

The DosDisk 'kit' is assembled (no pun intended) by writing several custom subroutines into a 'standard' overlay. The routines that need to be written are: Validate, Install, Uninstall, and (optionally) CRT status line routines. The optional CRT routines place and remove indications on the CRT's status line (if one exists) noting whether DosDisk is installed. A sample overlay is provided that is well written and documented. This sample even includes test stubs to allow each of the overlay routines to be tested before merging the overlay into DosDisk.

For your system to work with DosDisk it must support the following physical characteristics:

```
a DSDD 48 tpi 40 track drive
9 512 byte physical sectors, no skew
```

and the BIOS must support the following logical characteristics:

```
Non-Cylinder Mode
    2 tracks/Cylinder
    Even tracks on side 0, odd on side 1
Non-Inverted Data
Allocation Vector of at least 48 Bytes
Check Directory Vector of at least 28 Bytes
```

Your life will be made much easier if your BIOS also supports the overloading of disk parameter information for foreign formats.

A brief note on non-cylinder disk access. Technically speaking, a cylinder is considered to be both sides of a disk at any given head position. When accessing a disk as a cylinder, the head is stepped to a given track (e.g., track 3), then the data is read first from side 0, then from side 1 based upon sector number. When accessing the disk using non-cylinder mode, each side of the disk is treated independently. I know of two methods for doing this (and there are probably more).

The first, and most common, is the one required by DosDisk. Using this technique a 40 track disk is accessed as if it had 80 tracks. The low order bit of the track number is used as the side select thereby placing all the odd tracks on side 1 and the even tracks on side 0. Kaypros and the SB180 are two machines that immediately come to mind as using this format.

The second (used by the Superbrain) also treats the disk as if it had 80 tracks. However, with this format all of side 0 is accessed first, as tracks 0-39. Then side 1 is accessed as tracks 40-79. As Murphy would have it, that was the way my system worked. It was

BIOS butchering time. Because in the many years that I've had this system I've never needed the Superbrain format, I elected to switch the non-cylinder mode of my BIOS to the first method that I described above. Surprisingly, that change was really painless! So with that capability added to my BIOS, it was on to the DosDisk Overlay.

I like to start simply, so the first routine written was the validation routine. This routine just looks at the version number that I've coded into my BIOS right after the jump table. If it finds the incorrect version it returns with the error flag set. (I need to implement this check on some of the other programs that have 'hooks' into the BIOS, such as my Format Program.)

Next was the Install routine. My BIOS made this very easy to code. All I needed to do was copy the DosDisk DPB (disk parameter block) into a 'staging area' in my BIOS, set the 'foreign format' flag, then select the logical drive that DosDisk is to be installed on. This select causes my BIOS to move the 'foreign format' information from the 'staging area' into the DPB for that drive. If you don't have this capability, you will need to do a 'select' to get the address of your disk parameter block, copy the existing DPB into a data area provided by DosDisk, then move the DosDisk DPB into place. (Be sure you re-select the disk that was originally selected or you may have some problems.)

Finally, the Uninstall for my system was the easiest of all to create. I simply had to clear a 'Disk Logged' flag for the DosDisk drive. My BIOS then restores the 'native' DPB on the next select for that drive. On other systems it would be necessary to copy the 'native' DPB from DosDisk's internal data area back to the DPB buffer for that drive.

Since my terminal doesn't support a 25th status line, I didn't bother to implement either of the terminal routines.

If you use the sample overlay, you will find that it includes several useful routines conditionally assembled to aid in debugging. But, since my implementation was so simple, I elected to merge the overlay into the DosDisk COM file, load it with Z8E (my favorite debugger), then use Z8E to simulate calls to each of the overlay routines (before allowing DosDisk to run).

In summary, DosDisk has been an incredibly useful utility, paying for itself many times over in convenience. If you are fortunate enough to be able to get a preinstalled version, you're home free. But, don't let fear of installation hold you back. If you have had assembly language programming experience and know your BIOS, a couple hours of work will have DosDisk up and running. If you happen to be using a Computime S100 system, contact me and I'll provide my changes. •

Mysteries of PC Floppy Disks
(Continued from page 18)

all . . . at least, on floppy drives.

An Aside

Those of you who have Atari STs with single-sided drives might be interested in this single-sentence upgrade to double-sided drives: 1. Open the box; 2. Disconnect and remove the old drive; 3. Jumper the new drive for DS1 (not DS2 as the clones do it); 4. Connect the drive up; 5. Close the box (you might want to cut some holes in it if the light is in a different place, etc.).

Floppies of the Near Future

There are some new developments in store in the floppy disk arena. Unlike hard disks, the floppy disk market seems to innovate at a glacial pace. This is because of the need for compatibility and low price.

Zenith's MinisPort laptop uses 2 inch floppies. These drives use a 720K byte format exactly like the 720K 3.5 inch format.

Several manufacturers seem to be working on compatible 3.8 megabyte 3.5 inch drives. It appears that these drives use a higher track density, with more accurate stepping, rather than RLL or any other more exotic technology. Thus, they are compatible with older formats.

Other companies are playing with vertical recording. Another company, Insite Peripherals, increases track density by burning concentric circles on the media to delineate tracks and thereby get more on the disk. Their drive stores 20 megabytes on an expensive floppy.

The most promising developments for further in the future are the RAM cards. These devices are finding increasing use in various portable devices. Problems at this point are high cost, the lack of a standard data encoding format, and finding connectors that don't wear out and work well under a variety of humidity and temperature conditions.

Of course, there are WORM drives, laser cards, removable hard disks, and writable CD-ROMs. But in each case, either the drive or the media, or both, is too expensive for mainstream use.

Low-cost floppy drives were greatly responsible for the PC revolution, and removable, exchangeable, and distributable media will always be required for the future. So, it looks like floppies are here to stay, be they whatever size.

Oh, and the earth-shattering conclusion? Floppy disks were actually not invented by IBM, as you may have heard, but were transported to Earth by space aliens who had been in psychic contact with Elvis Presley. Actually, they were demonstrated by General Electric at the 1939 World's Fair. Thomas Edison described them in 1896. Nikola Tesla had the idea first, but his models disappeared into the Bermuda Triangle. Amelia Earhart knew the whole horrible story. Elvis took it to his grave. Micheal Jackson is buying the rights. You read it right here. It's all true – except this paragraph. •

References:

The PC Sourcebook, by Tom Hogan, Microsoft Press, 1988

Technical Reference - Personal Computer, anonymous, IBM Corporation, 1984

Programmer's Guide to the IBM PC, by Peter Norton, Microsoft Press, 1985 (aka the Pink Shirt Book)

Tandy 4000 Technical Reference Manual, anonymous, Tandy Corporation, 1987

User Disk

The code from *Multitasking in Forth, Animation with Turbo C, Mysteries of PC Floppy Disks*, and *Real Computing* is available on a 5.25" 360K or 3.5" 720K PC format disk for $10 postpaid in the U.S.

Advanced CP/M

ZMATE—The Z-System Programmer's Editor and Lookup and Dispatch

by Bridger Mitchell

ZMATE

Exciting news at press time! Jay Sage and I are completing the arrangements to offer a new, Z-System version of the superb MATE editor, which Michael Aronson created more than a decade ago for use by members of his research group in the Harvard Physics Department. They referred to it simply as Mike Aronson's Text Editor, and the initials MATE became its name. When Phoenix Software picked it up as a commercial product, they added their 'P' trademark, and it became PMATE. It also grew into an MS-DOS version (which is Jay Sage's DOS writing tool).

Jay and I had long wanted to bring it up to date. Our objectives included: recognize drive-user and named-directory file references, install automatically on a Z-System by getting screen codes from the TCAP, offer split-screen display, enhance several commands, and fix a few lingering bugs. Mike Aronson was agreeable in principle, but, alas, the source code could not be located. Fortunately, I succeeded in fully disassembling the editor. Then I converted it to Z80 code and gradually added the new features. In its new incarnation we call it ZMATE.

ZMATE is a macro-driven editor. This means that virtually everything the editor does can be programmed, and reprogrammed, by the user. The implications are far-reaching. First, you can automate both simple, repetitive operations and very complex editing tasks. The macros you create can be saved and reused. You can assign them to specific keys or sequences of keys, or you can invoke them as single-character commands. Second, you can "rebind" the keyboard commands to whatever keys you prefer to use, which makes learning ZMATE and switching to it from another editor go more smoothly.

ZMATE uses a main text buffer, which supports virtual memory (the size of the file being edited is limited by disk space and not by memory) plus 10 supplementary, numbered buffers. The numbered buffers can be used to hold chunks of text, or other files, or macros. ZMATE's new split-screen mode lets you see, simultaneously, text from two different buffers or from two parts of the same buffer. Two-window capability is extremely handy—for keeping track of notes while writing, for viewing a list of equates when coding, and for developing and testing macros.

Final arrangements are being worked out at press time, but we expect to be able to make ZMATE available at the incredible price of $50 (the last version of PMATE from Phoenix was $225!). Write to Plu*Perfect Systems or Sage Microsystems East for further information. And watch future TCJ columns for some nifty ZMATE macros!

Lookup and Dispatch

In last issue's Advanced CP/M we considered several methods of efficiently passing parameters to a subroutine. This time our topic is multi-way branches—the efficient dispatching to one of several possible routines.

In-line Comparison.

When there are just a few branches, a series of in-line comparisons is the natural choice.

```
        cp      'a'
        jp      z,a_addr
        cp      'b'
        jp      z,b_addr
        ; ...
otherwise:
```

This is quickly written and reads clearly when there are just a couple of alternatives.

In a high-level language, this might be written:

```
        case    'a':
                a_addr();
                break;
        case    'b':
                b_addr():
                break;  ...
        default:
                otherwise();
```

The Z80 code can be shortened a bit if the keys are consecutive values. Subtract the smallest key to convert the value to small integers:

---

*Bridger Mitchell is a co-founder of Plu*Perfect Systems. He's the author of the widely used DateStamper (an automatic, portable file time stamping system for CP/M 2.2); Backgrounder (for Kaypros); BackGrounder ii, a windowing task-switching system for Z80 CP/M 2.2 systems; JetFind, a high-speed string-search utility; DosDisk, an MS-DOS disk emulator that lets CP/M systems use pc disks without file copying; and most recently Z3PLUS, the ZCPR version 3.4 system for CP/M Plus computers.*

*Bridger can be reached at Plu*Perfect Systems, 410 23rd St., Santa Monica CA 90402, or at (213)-393-6105 (evenings).*

```
        sub     'a'     ; subtract smallest key
        jp      z,a_addr
        dec     a
        jp      z,b_addr
        ;...
otherwise:
```

Key/Address Tabulation

When there are more than half a dozen branches, it becomes worthwhile to put the cases into a table. A straightforward, popular data structure is:

```
TERMINATOR      equ     0FFh
;
tbl:    db      a_key
        dw      a_address
        db      b_key
        dw      b_address
        ...
        db      TERMINATOR
```

The real benefit of using a data structure over in-line code is the readability and maintainability of the code. If we use a simple macro:

```
.cmd    macro   key,addr
        db      key
        dw      addr
        endm
```

then the same table can be written very clearly as:

```
tbl:    .cmd    'a',a_addr
        .cmd    'b',b_addr
```

As the program evolves, changes are immediately apparent. We add a new key and its address to the table and provide its matching routine, or change the key used for an existing routine to better represent it mnemonically.

In each example that follows, we will use a lookup routine that is used in this general form:

```
load registers with parameters
call lookup routine
if not found, jump to error
else jump to address of matching routine
```

One lookup routine for this data structure is shown in Figure 1. It steps through the table, comparing the A register value with each tablulated key. If it matches, it loads the next two bytes as the dispatch address. If not, it skips past those bytes and tests the next key.

Block-compare Dispatching

The Z80 is renowned for its block-move instructions (LDIR, LDDR). We can exploit one of the similar block-compare instructions to make a more efficient lookup routine.

The essential idea is to separate the keys and addresses into two, parallel tables:

```
keytbl: db      'a'
        db      'b'
        ...
addrtbl:
        dw      a_addr
        dw      b_addr
        ...
```

We can search the keys by setting HL to point to the keytable and BC to the number of tabulated keys. The CPIR instruction will then scan the full table, returning with the Z flag set if a match is encountered, and the BC with the number of keys not yet checked.

Figure 1. Lookup Routine for Key/Address Table

```
; Enter:
;       a = index
;       hl -> table, terminated by 0FFh
; Exit:
;       Z if error, else hl = dispatch address

lookup: ld      b,(hl)          ; if FF terminator
        inc     b
        ret     z               ; ..exit Z
        cpi                     ; check key for match
        jr      z,match
        inc     hl              ; skip over address word
        inc     hl
        jr      lookup
match:  or      0ffh            ; set NZ
        ld      a,(hl)          ; get address
        inc     hl
        ld      h,(hl)
        ld      l,a
        ret
```

Figure 2 -- Lookup using Block-Compare Instruction

```
;
; Enter:  a = index
;         hl -> end of key table
;         bc = # entries
;         de -> start of address table for keys
; Exit:   Z if match, hl = dispatch address
;
lookup: cpdr            ; compare and repeat
        ret     nz      ; ..not found
        ex      de,hl   ; point at address table
        add     hl,bc   ; index into it
        add     hl,bc
        ld      a,(hl)  ; get the address
        inc     hl
        ld      h,(hl)
        ld      l,a
        ret
;
```

Figure 2 is a lookup routine that uses these techniques. Actually, it's more compact to start at the end of the table and use the CPDR instruction. That way, the final BC value can be used to index into the address table.

It seems, though, that in splitting the keys and addresses we've been forced to use an inelegant and error-prone data structure. But we can design a simple macro to get around this:

```
.cmd    macro   key,addr
  cseg
        db      key
  dseg
        dw      addr
  cseg
        endm
```

My personal convention is to label macros with a leading "dot" (thus, ".cmd"). This helps to set them apart from other names in a stream of source statements. In this macro, we make use of two different relocation bases—the regular code (CSEG) base for the key table, and the data (DSEG) base for the address table. We can then again write the table as

```
tbl:    .cmd    'a',a_addr
        .cmd    'b',b_addr
```

In some cases, though, we may want to keep the two tables together in the same relocation base. With rather more effort, we can develop a pair of

```
        Figure 3 -- Key/Address Table Macros for Code Segment
;
; usage:
;
;       .precmd n,keytbl,addrtbl        ; number of keys, labels
;       .cmd    'A',a_addr              ; first key, its address
;       .cmd    'B',b_addr              ; etc.
;       .cmd    %CNTL_A,cntl_a_addr     ; use '%label' for non-printing key
;       .postcmd                        ; check for correct length

; Define the origins for the key and address tables.
; Define the (global) labels for those tables.
;
.PRECMD macro   n,keylabel,addrlabel
;; note -- these are all global labels
???keyorg       defl    $
???addrorg      defl    $+n
???keycnt defl  n
;
keylabel equ    $
keylabel&end equ $+n-1
addrlabel equ   $+n
        endm

; Create an entry in both tables.
;
.CMD    macro   key,addr
;; test for missing parameters, force an error if so
  if nul key
.error MISSING PARAMETER!
  exitm
  endif
  if nul addr
.error MISSING PARAMETER!
  exitm
  endif
;;
???keycnt defl  ???keycnt-1
  org   ???keyorg
        db      key
        ???keyorg       defl    ???keyorg+1
  org   ???addrorg
        dw      addr
        ???addrorg      defl    ???addrorg+2
        endm    ;; .cmd
;
; Check for correct table length parameter
.POSTCMD        macro
  if    ???keycnt
.error  BAD TABLE COUNT
  endif
        endm

                Figure 4 -- Lookup With Implicit Keys
;
; Enter:
;       a = index
;       b = min. index, c = max. index+1
;       hl -> table of addresses
; Exit:
;       CY if error
;       else hl = dispatch address
;
lookup:         cp      b               ; if index <  min. value
                ret     c               ; ..error
                cp      c               ; or if > max. value
                ccf
                ret     c               ; error
                sub     b               ; make relative index
                add     a,a             ; double rel. index
                add     a,l
                jr      nc,lookup1
                inc     h
lookup1:        ld      a,(hl)
                inc     hl
                ld      h,(hl)
                ld      l,a
                or      a               ; clear cy
                ret
```

macros to do the same thing entirely in the code relocation base. They are shown in Figure 3. The ".precmd" macro is invoked first, with the number of cases (N) that will be used in the table, and the labels for the key and address tables. This is followed by N ".cmd" macros, as above. Finally, the ".postcmd" macro is invoked, to check that there were actually N table entries declared.

Writing assembler macros is always a tricky business. It took me several iterations to get this one working! So it's good practice to document its use, and to build in error-checking.

One disadvantage of this macro pair is that N, the number of table entries, must be a hand-calculated constant—it must be a known value when the .precmd macro is invoked. You can't define N as the number of .cmd statements, because that value isn't available to the assembler until it gets to the end of the table on its first assembly pass. The best I could come up with is the third ".postcmd" macro to check that the precalculated value actually agrees with the number of macro invocations.

The contortions required to write this macro pair illustrate the need for a more powerful macro language. What we would like to have is a way to postpone using the address parameter of all invocations of the .cmd macro until all of the parameters in the key table have been constructed. The old TDL assembler has such features. Using it, one can declare temporary variables. In this application we would use them to hold the address values from successive .cmd macros, and then in a different ".postcmd" macro define all of the address words.

Exhaustive Case Table

When almost all of the keys in a range of values are to be used, the keys themselves can be omitted. In this case the addresses in the address table serve as implicit keys. No searching is required; the correct table location is calculated directly from the key value. Suppose, for example, we have keys 'A' through 'P'.

The address table is then:

```
addrtbl:        dw      A_addr
                dw      B_addr
                ...
                dw      P_addr
```

Note that the addresses must now be in "alphabetical" order, with no gaps. If there are just a couple of keys in the sequence that aren't in use, you can use a the address of a dummy (or error) routine for them.

A lookup routine for the exhaustive case table is shown in Figure 4.

Discussion

A lookup function can be reused for several different tables throughout a program. It thus makes a good method for a menu, or command-driven application in which there is some type of

hierarchy. It also finds heavy use in major tools—assemblers, linkers, database managers, and so forth.

Whatever method is used, it is essential to have a range check and an error routine to handle illegal key values.

Despite the uniformity of these examples, the addresses in the lookup routines needn't be subroutine entries. The same routine can be used to locate a message string corresponding to a message code key; the address of a further data structure indexed by a key byte; and for other key/pointer uses.

Still another application of lookup routines is translation from one key to its replacement value. Here, the second table would be a table of single bytes, rather than words. Straightforward modifications to the routines used here can serve that purpose. •

Plu*Perfect Systems == World-Class Software

BackGrounder ii **$75**
Task-switching ZCPR34. Run 2 programs, cut/paste screen data. Use calculator, notepad, screendump, directory in background. CP/M 2.2 only. Upgrade licensed version for $20.

Z-System **$69.95**
The renowned Z-System command processor (ZCPR v 3.4) and companion utilities. Dynamically change memory use. Installs automatically
Order **Z3PLUS** for CP/M Plus, or **NZ-COM** for CP/M 2.2.

ZMATE **$50**
New Z-System version of renowned PMATE macro editor with split-screen mode for two-window viewing of one or more files. Extremely powerful and versatile macro capability lets you automate repetitive or complex editing tasks, making it the ultimate programmer's editor. Macros can be saved for reuse and also assigned to keys. Editing keys can be reconfigured for personal style. Supports drive/user and named-directory file references. Auto-installs on Z systems. Z-80 only. Supplied with user manual and sample macro files.

PluPerfect Writer **$35**
Powerful text and program editor with EMACS-style features. Edit files up to 200K. Use up to 8 files at one time, with split-screen view. Short, text-oriented commands for fast touch-typing: move and delete by character, word, sentence, paragraph, plus rapid insert/delete/copy and search. Built-in file directory, disk change, space on disk. New release of our original upgrade to Perfect Writer 1.20, now for all Z80 computers. On-disk documentation only.

ZSDOS ..... **$75, for ZRDOS users just $60**
State-of-the-art operating system. Built-in file DateStamping. Fast hard-disk warmboots. Menu-guided installation. Enhanced time and date utilities. CP/M 2.2 only.

DosDisk ..... **$30 - $45**

Use MS-DOS disks without copying files. Subdirectories too. Kaypro w/TurboRom, Kaypro w/KayPLUS, MD3, MD11, Xerox 820-I w/Plus 2, ON!, C128 w/1571 -- $30. SB180 w/XBIOS -- $35. Kit -- $45. Kit requires assembly language expertise and BIOS source code.

MULTICPY **$45**
Fast format and copy 90+ 5.25" disk formats. Use disks in foreign formats. Includes DosDisk. Requires Kaypro w/TurboRom.

JetFind **$50**
Fastest possible text search, even in LBR, squeezed, crunched files. Also output to file or printer. Regular expressions.

To order: Specify product, operating system, computer, 5 1/4" disk format. Enclose **check**, adding $3 shipping ($5 foreign) + 6.5% tax in CA. Enclose invoice if upgrading BGii or ZRDOS.

Plu*Perfect Systems
410 23rd St.
Santa Monica, CA 90402
(213)-393-6105 (eves.)

BackGrounder ii ©, **DosDisk** ©, **Z3PLUS** ©, **PluPerfect Writer** ©, **JetFind** © Copyright 1986-88 by Bridger Mitchell.

Real Computing

The National Semiconductor NS320XX

by Richard Rodman

Benchmarks

There are only two kinds of benchmarks: The ones you do yourself—and lies.

I tested the 32532 Designer's Kit using the Prendeville C compiler and the Internet assembler by Bruce Culbertson. For comparison, I used Microsoft C large model running under DOS on a variety of machines. The numbers shown in Figure 1 are in Dhrystones per second; that is, 50000 divided by seconds.

For comparison, the VAX 11/780 as given in the Dhrystone source file is listed at some various numbers around 1500.

Editor's Note: The DHRYSTON.C 28.6KB file with the results and C source code is available on disk, or hardcopy for an addressed #10 envelope with 45 cents postage.

Now everyone knows that the Dhrystone benchmark is as much or more a test of compiler technology than of CPU integer performance. This has been interpreted by many as a cue to "hand-optimize" the compiler output or use other exotic technology to claim astronomical benchmark figures. The problem here is that *your* programs cannot benefit from these techniques. The only benchmarks that are usable are those that are computed using standard off-the-shelf compilers.

Intel 80x86 benchmarks should also be always done in large model. Small-model performance is only an indicator of how trivial programs will run.

Anyway, I expected the 32532, running with no wait states using 35ns static RAMs, would do better. So, I am fiddling with the compiler to see if improvements can be made.

A surprising result is the quick performance of DOS programs in the OS/2 DOS compatibility box, for which I have no explanation. Incidentally, some people have suspected me of being a closet OS/2 fan. That is not true. I am a *non-closet* OS/2 fan.

Fast Packet

Low-overhead network protocols are beginning to move through the standards bodies. The big talk in the packet-switch world is a new protocol called *Frame Relay*, which basically is a modified version of level 2 of X.25. Rather than a step-wise acceptance of responsibility for data integrity at each node, error checking is only performed at the end nodes, and in the event of error, retransmission is on an end-to-end basis.

The thinking behind this new mechanism is that the all-digital fiber networks of today are virtually error-free, and there isn't as much need for so much protocol. By moving error checking to the circuit ends, there is much less work to do in moving the data. In fact, once the address portion has been received, the switch can begin transmitting the outbound packet while the inbound packet is still being received. This will have the effect of greatly reducing cross-network delays.

ANSI, CCITT and the other standards bodies generally take it upon themselves to add as much baggage as they can to any proposed standard. That's why you get so little actual performance compared to your theoretical value.

But the world seems to go through cycles of ornamentation and simplicity, of embellishment and austerity. The swing today is to the smooth, the clean, the simple, straightforward, and honest. The challenge is to never mistake simplicity with being simplistic, forcing simplicity upon something which is really complex. As Albert Einstein once said, "It should be as simple as possible, but no simpler." To which I add, as complex as necessary, but no more complex. Programmers take note.

More Bones to Pick, Axes to Grind

Intel is currently trying to sell everyone on a benchmark which they call Rhealstone. This benchmark is supposed to measure embedded-system performance, but in fact only measures interrupt response and context-switch time. While the NS32 chips do in fact have about the best context-switch time in the industry, I think it's intellectually dishonest on Intel's part to say that embedded system performance is predominantly measured by these factors, because their register-poor processors do well in these areas, but are computationally deficient for the same reason.

Then there's another company that's actually managed to establish a "standard," actually cited by some government RFPs, that any supplied operating system must be totally compatible with the current version of their proprietary operating system (it has some Greek-sounding name).

This kind of behavior might be defensible if these "standards" had any technical merit, but they don't. They're like GM getting the government to mandate that all

```
32532, 25MHz, Designer's Kit, C16 . . . . . 7530
80286, 10MHz, AST Premium 286, DOS 3.3, MSC 5.1 . . . . 1786
80386, 16MHz, ALR-386, DOS 3.3, MSC 5.1 . . . . . 1852
80386, 16MHz, Tandy 4000, OS/2 1.2, MSC 5.1 . . . . . 1742
80386, 16MHz, Tandy 4000, OS/2 1.2 DOS box, MSC 5.1 . . 2252
```

Figure 1: Dhrystone results.

cars must use Corvair transaxles (Corvair is a trademark of General Motors Laboratories).

Then, even more laughable, in a sad sort of way, is the fact that a college student is being prosecuted because he unleashed a "worm" which entered government computers. It seems that he *overestimated* the "security" of the network to which his college's computers were connected. Using the word "secure" in connection with this "standard" operating system is an oxymoron indeed. Remember the old saw about "if builders built buildings the way that programmers wrote programs..."?

Rumor City

National will soon be making available an AT-form-factor motherboard using the 32GX32 chip. This motherboard will cost around $2600. For applications requiring an MMU, the 32532 may be substituted for the 32GX32. It's unfortunate that it costs so much, but big semiconductor houses just never seem to be able to produce affordable boards or systems.

The rumors have it that a whole series of motherboards is coming. It sounds fishy to me—after all, people have been asking for it for years. National has announced a new IC that has the floppy controller, printer and serial interfaces, game controller interface, and decoding logic all on one chip.

Isn't it strange, though, that the parts places that cater to experimenters seldom carry any newer devices. Maybe they're just overstocked with 8085s, 6820s, and Z80-DMAs. The big parts companies still don't have the imagination to see that the experimenters and hobbyists who pay real money for these parts and have the determination to make something out of them are the CEOs of the future. Do these folks a little favor, and they'll repay you a million times. Do them a bad turn, and you'll never sell them a free lunch.

In fact, if I was running National Semiconductor, I'd practically give the boards away. (I hope someone at National reads this!)

Language of the Future—Here Today

If you look back at the magazines from the opening years of the real PC revolution, 1976 and 1977, you'll remember that, once Tiny Basic was readily available, there was universal assent as to what the language of the future would surely be. This language was interactive, giving instantaneous feedback, with unmatched power and brevity of expression. It was the ultimate interpreted language, compressing complex operations into single character tokens. Everyone knew it, and there was no stopping it. It was APL.

Today, while multitudes struggle with the thorny complexity of C, now ANSIfied with an ever-tightening noose of stricter and stricter syntactical rules (how about this cast: "(void (*)(void))"?), a small, soft-spoken minority has arrived at the conviction that the next generation must rise above this cacophony of petty bit-twiddling and move mankind into the *real* information age. These folks are distributing an APL interpreter for PCs at cost, hoping to reach the younger generation. They are even working on a version for the lowly Commodore 64.

They will answer the phone: "I APL - do you?". Alas, I must admit, I do not. I used Vanguard APL on a CP/M system with a video board—I customized the character generator myself—but, while I agree that the math symbols are sure powerful, I can't remember what the darn things mean. Kids have these spongelike brains, though, and it seems like they can pick anything up. ●

Where to call or write

BBS: 703-330-9049 (Richard Rodman)

I APL, do you?
Edward Cherlin
APL News
6611 Linville Drive
Weed CA 96094

Animation with Turbo C
(Continued from page 10)

Bringing It All Together

There is insufficient space available to wrap this topic up here. Now that we have most of the basics down and have done some experimenting we are anxious to put our new learning to use with some exciting screen action programs, right?

Part two will continue where we are leaving off—that is, we will start by firing shells from the tank at several different angles. To do this we begin with creating and saving the shell image. We then have to establish departure coordinates for each angle of the shell, we can use the 'A','S','D',and ,'F' keys for this. Then we'll need to detect the screen sides and top to know when to stop the shell travel. And when the shell comes in contact with a target that needs detection and appropriate follow up action such as replacing the shell with a burst and a bit of sound, making the target disappear, and providing a scoring update.

Two groups of three aircraft fly across the screen in the complete game. The two groups fly in opposite directions. When all three of a group have been shot down a new group reappears elsewhere on the screen. One group drops bombs on the tank, the other fires missiles. There is a lot of interesting program logic involved and it can be tricky. Something to look forward to.

Summary

In this segment of Turbo C graphics programming we have extended our knowledge of viewports and learned we can perform fairly decent animation with the putimage() function. We have also discovered how to use the two text functions, outtext() and outtextxy() inside a viewport and employment of sprintf() for formatted text. In all this we are also overcoming difficulties the authors of manuals and various texts on graphics allow us to discover for ourselves. ●

User Disks

The code from *Multitasking in Forth, Animation with Turbo C, Mysteries of PC Floppy Disks*, and *Real Computing*, is available on a 5.25" 360K or 3.5" 720K PC format disk for $10 postpaid in the U.S.

The complete installation package of Metal 0.6, including C compiler, assembler, linker, make utility, ROM monitor, host program, and various utilities, is available on a 3.5" 1.44 MB, 3.5" 720K, or 5.25" 360K PC format disks for $13 postpaid in the U.S.

Forth Column

Strings

by Dave Weinstein

This (somewhat belated) column will be somewhat more diverse than previous columns, and will also be the first of a two-part series on strings. First, a discussion of some of the latest activities in the Forth community.

Those using Unix based systems (or some sort of non-standard equipment with a Unix compatible C and C library) will be interested in a new (and extremely impressive) Forth from Mikael Patel. TILE Forth is a large 32-bit Forth, complete with C source code, documentation, and all of the features that users of so-called "fat" Forths have come to expect... Moreover, TILE is the platform upon which a complete Object Oriented Forth system (FORTHTalk) is being built. The environment is the traditional single line Forth interpreter. Because TILE is designed for a multitasking environment there is no editor built in (but public domain Forth editors exist in abundance). For those who use EMACS, the TILE package includes a LISP file to configure a Forth interactive environment. The fact that such an intensive and well developed package (and other similar programs, such as F-PC) are freely released to the community at large is one of the strengths of the Forth community. Unfortunately, it also hinders acceptance in the industry at large. Many managers are leery of unsupported or "free" programming languages (although the support of gcc and g++ have done much to allay these fears), and there is not as much of an incentive to write professional Forth interpreters or true compilers when the market is not only small, but is also dominated by free competitors. (This is probably about the time I should step off of my soapbox)

The other big piece of news is the RTX 2001 giveaway by Harris Corporation. Harris is running a "Real Time Design Contest" (well, to be more precise the contest is being co-sponsored by *Embedded Systems Programming* and Harris). Essentially, interested people submit design specifications by April 16, 1990. If the design is deemed reasonable by the sponsors, an RTX 2001A development kit is given to the designer. The finished designs will be evaluated, with multiple prizes including a $10,000 grand prize, two $2,000 prizes for best hardware and best software respectively, as well as $500 prizes for the runners up in the aforementioned categories. The contest (as described by Harris) will include a giveaway of up to 1000 systems. When I spoke to Harris to find out where my application form was, I discovered that they have been swamped with interested callers. (Whether they'll up the number of systems to be given away is another matter). Finally, winning designs will be shown at the second annual Embedded Systems Conference (September 25-28, 1990) in Burlingame, California.

Strings and String Handlers

There are two common ways of expressing strings. The first, or Pascal style, is with a length field preceding the string. The second, or C style (also called ASCIIZ) is with a null (ASCII 0) character as a terminator. Each of these implementation methods has its advantages. Pascal style strings make determining the length of a string trivial, and allow possible shortcircuiting of string comparisons (if they are of different lengths, don't even bother with the comparison). The primary advantage of the C implementation (other than the fact that it is the more common, and therefore more useful when dealing with most operating systems) is that it places no limitations on the size of the string. In the true spirit of flexibility (or was that waffling), we'll just go ahead and use both.

This column will sketch out possible implementation methods for the strings, and design the object shell (in other words, we'll know how to deal with the strings, but we won't actually implement the code until next issue). There are a few reasons for this (other than time and space considerations). First, one of the biggest advantages of an object oriented approach is that it allows us to discount the implementation completely. If written properly, we should be able to change the implementation of a class and have all code which uses objects of that close continue to work. Secondly, next issue will contain at the very least two implementations of the strings, one with a maximum length string, and the other, using a memory management object, with variable length strings. There would not be enough room to combine that code with the design details, so we'll split them.

Valid Operations on Strings

Before even starting laying code for strings, we need a list of valid operations on strings, and of their functionality. A list of string operations and brief descriptions can be found in Figure one.

String Primitives

The first of these operations is, obviously enough, Create. In an object oriented implementation, this function will be performed as part of the initialization process (the object oriented package we are using supports auto-initialization of objects). This is perhaps the most implementation dependent of the operations, because it is responsible for putting the object in a state where all of the other operations are valid. Because of this, it won't really be covered in detail until next issue, when I start talking about implementations.

The second operation is Clear. Simply enough, it sets a given string to be the null string.

The third, fourth, fifth, sixth, and seventh operations all are information accessors. They do not change the string at all. Length, simply enough, returns the length of a given string, Error? returns a flag, which is true if a previous operation has set the error state, and Extract. Extract returns the nth character of a given string, with the error flag set if n is greater than the length of the string. The last of the accessor operations, Full? and Empty?, return booleans, and are true if the string in question is full or empty respectively.

The last two operations actually change the string they act upon. ClearError, simply enough, resets the error flag in the String object. Concatenate, given a character, appends that character to the string. The objects internal error flag is set if the string was already full.

These are the string primitives. They are the only operations which need to be

implemented as methods in the String object, because all other valid string operations can be written in terms of these. Anything else which we need can be coded as standard Forth words. The resulting code would be highly inefficient, but it could be done. The remaining operations will in fact be coded as part of the objects for efficiencies sake, but the important idea is that we don't need to do so.

Other String Operations

The other operations we need for strings (at least to make them useful) are slightly more complicated. Again referring to Figure 1, the operations we will be implementing are...

Compare. Simply enough, this operation compares the results of two strings, returning true if they are equal. A more complicated version of this operation (an exercise for the reader?) would be to modify it to have the same functionality as strcmp () in C. (For non C programmers, it would return -1 if the first string is "less than" the second, 0 if they are identical, and +1 if the first string is "greater than" the second).

Append. Given to strings as arguments (it is a method acting upon the target string), this operation sets the target string equal to the second string appended to the first. Again, if we were interested in a minimal object definition, this could be coded as a loop of Concatenates and Extracts. It would just be mindbogglingly slow.

ExtractString. Rather than merely extracting a single character, operation extracts an entire substring (length and position are passed to it along with the string from which the substring should be extracted).

PrintString. This operation merely prints the string. Of all of the operations, this is the one most likely to be redefined by sub-classes (so we have PrinterString class and WindowString class and TitleString class, and so on).

You'll note that there isn't an operation given for setting strings from inside of code. This sort of operation is inherently non-portable. So rather than design semantics for it here (which will be unusable for various people, depending on the Forth implementation being used), the implementation article (next issue) will cover the art of getting information into the strings other than one character at a time.

Final Thoughts

This has been a short column. Next issue on the other hand, will be larger than normal (it was a trade off, and it takes more time to discuss actual coding involved than it does the theory and design). I still haven't heard what sorts of columns you'd like to see? (I'm running out of ideas and running out of them fast). •

```
        Figure 1: String Operations

        Primary Operations

Create      ( string -- )
        PreConditions: None
        PostConditions: String intialized

Clear       ( string -- )
        PreConditions: String initialized
        PostConditions: String reset to null string

Length      ( string -- len )
        PreConditions: String initialized
        PostConditions: No change.

Error?      ( string -- f )
        PreConditions: String initialized
        PostConditions: No change.

Full?       ( string -- f )
        PreConditions: String initialized
        PostConditions: No change.

Empty?      ( string -- f)
        PreConditions: String initialized
        PostConditions: No change.

Extract     ( position string -- char )
        PreConditions: String initialized
        PostConditions: Error flag set if position > length.

ClearError  ( string -- )
        PreConditions: String initialized
        PostConditions: Error flag cleared

Concatenate ( char string -- )
        PreConditions: String initialized
        PostConditions: Character added to end of string. Length incremented
                        Error flag set if the string was full.

                Secondary Operations

Compare      ( string-1 string2 -- f )
        PreConditions: Strings initialized
        PostConditions: No change

Append       ( string-1 string-2 new-string -- )
        PreConditions: Strings initialized
        PostConditions: string-1 and string-2 unchanged. new-string set
                        to string-1 + string-2. Error flag in new-string
                        set if the lengths of string-1 and string-2
                        combined are greater than the maximum allowed

ExtractString ( length position string-1 string-2 -- )
        PreConditions: Strings initialized
        PostConditions: string-2 set to a length sized string which starts
                        at position in string-1. Error flag in string-2
                        set if length is greater than maximum allowed, if
                        position is illegal, or if position + length is
                        greater than the length of string-1.

PrintString    ( string -- )
        PreConditions: String initialized
        PostConditions: No change.
```

If You Don't Contribute Anything

Then Don't Expect Anything

TCJ is User Supported

The Z-System Corner

By Jay Sage

For this issue (and the next) I am going to indulge myself and write about about something that I enjoy, even though, strictly speaking, it has nothing to do with Z-System. This subject is MEX-Plus, the most advanced telecommunications package available for CP/M computers. I suppose I could argue that there is a philosophical or spiritual connection, since MEX-Plus allows the user do for telecommunications many of the same things that Z-System allows one to do with the operating system, namely automate. In particular, I will be describing MEX-Plus's scripting capabilities, which are similar in some ways to alias and ARUNZ scripts in Z-System.

I had been hoping that a regular MEX column would develop here in TCJ, but that hasn't happened. Two of the people who might have gotten it going, Bruce Morgen and Rick Charnes, are now employed in the MS-DOS industry, and, after programming all day professionally, they don't seem to have as much energy left for hobby computing as they used to. Recently, David Goodenough, author of the QTERM telecommunications package, came over to my house and got QTERM running on my SB180 with its Wyse 50 terminal. I am enormously impressed with what David has accomplished with this program; it is rapidly developing many of the capabilities of MEX-Plus. I can envision much discussion in the future of both MEX-Plus and QTERM scripts. You can also expect an article or two about QTERM.

For this column I will give an overview of MEX's command structure, and next time I will describe in detail my suite of scripts for using PC-Pursuit as an example of what can be done using those commands. (I originally planned to cover both in one article, but after finishing the description of the commands, I was already at my limit.) Along the way I will try to point out some of the bugs and idiosyncrasies that users have discovered in MEX commands and suggest means to get around them. I hope that my examples will help give others ideas about how to make better use of MEX-Plus. Although regrettably I have had very few takers in the past, I again extend an invitation to readers to send me suggestions and examples for scripts.

Jay Sage has been an avid ZCPR proponent since the very first version appeared. He is best known as the author of the latest versions 3.3 and 3.4 of the ZCPR3 command processor and for his ARUNZ alias processor and ZFILER point-and-shoot shell.

When Echelon announced its plan to set up a network of remote access computer systems to support ZCPR3, Jay volunteered immediately. He has been running Z-Node #3 for more than five years and can be reached there electronically at 617-965-7259 (MABOS on PC-Pursuit, pw=DDT). He can also be reached by voice at 617-965-3552 (between 11pm and midnight is a good time to find him at home) or by mail at 1435 Centre St., Newton, MA 02159. Jay is now also the Z-System sysop for the GEnie CP/M Roundtable and can be contacted as JAY.SAGE via GEnie mail or chatted with live at the Wednesday real-time conferences (10pm Eastern time).

In real life, Jay is a physicist at MIT, where he tries to invent devices and circuits that use analog computation to solve problems in signal, image, and information processing. His recent interests include artificial neural networks and superconducting electronics. He can be reached at work via Internet as SAGE @LL.LL.MIT.EDU.

Overview of MEX Commands

The original CP/M telecommunication program MODEM7 and its derivatives, such as IMP, operate in two modes: terminal mode and command mode. Terminal mode is used for talking to the remote system, while command mode is used to control the local system. These programs have a relatively small set of commands falling principally into two classes: those required for file transfers and those related to the setup of the program.

In MEX-Plus the arsenal of commands is extended enormously, and anything that could be done from terminal mode can be done from command mode as well. (You would not want to run an interactive session this way, but it could be done.) There are so many commands that, though I will mention almost all of them, I will not be able to explain them all in detail. Rather, I hope to give you a general picture of the kinds of tools one has to work with in MEX-Plus. I will take up the commands in groups.

Setup Commands

Most of MEX's configuration is controlled by three commands: STAT, SET, and TSET. The STAT command works with more than 50 different options! All of them can display information about how the system is configured, and all but a few also allow the configuration to be changed. The MEX STAT parameters fall into four classes: switches, characters, values, and miscellaneous.

Switches have values of ON or OFF. One example is FILTER, which controls whether MEX will filter from the incoming modem stream any control characters other than tab, backspace, carriage return, and linefeed. If one wants to perform full screen operations, this filter must be off. Otherwise, the escape characters that initiate a screen control sequence will be swallowed by MEX.

Character STAT parameters take a single ASCII character as their value. An example is SEP, the multiple command separator (like Z-System, MEX-Plus allows multiple commands on a line). Value parameters take on numerical values. For example, PAGE sets the number of lines in a page on the screen, and CLOCK is set to the clock speed of your microprocessor chip so that timing delays will come out right. The miscellaneous STAT parameters mostly control the display of groups of information. "STAT ?" lists all the STAT parameters; "STAT VAL" shows the value parameters.

The SET command controls the modem setup. It is used for setting the baud rate, word length, number of stop bits, and parity mode. The TSET command was one I never used until working on this article. It controls special features related to the terminal

(what it does depends on the terminal you are using). With my Wyse 50, a TSET parameter can force all characters to be displayed in upper case characters or in highlighted video (that should give you some idea of why I have never used this command).

Operating System Commands

Another group of commands deals with the interface to the operating system.

Naturally, there has to be a way to get out of MEX and back to the operating system prompt. Have you ever been trapped inside a program, not knowing how to exit? Well, MEX author Ron Fowler must have had such a harrowing experience, because he has six (count 'em!) commands for getting out: BYE, EXIT, QUIT, CPM, DOS, and SYSTEM. SYSTEM must come from BASIC, which is the only CP/M program I ever became trapped inside. MEX-Plus is available in an MS-DOS version as well, and that is where the exit command DOS comes from. As a CP/M diehard, I take great pleasure, as you can imagine, in exiting from my DOS version using the command CPM (it doesn't help—I'm still in MS-DOS after I exit).

There are commands for doing operating system chores: DIR, REN, ERA (or, following MS-DOS, DEL), TYPE, and LOG (change drive/user). Some of these commands not only perform their functions; they also return information for use in a script file. DIR, for example, sets a special variable to the number of matching files found, and thus can be used to determine whether a particular file exists or not.

There are the commands KEY for associating strings with any ASCII key on the keyboard and PHONE for entering phone numbers into a dialing library. The key definitions and phone number library can be stored in disk files with the respective extensions KEY and PHN. The commands SAVE and LOAD write the data to and read the data from the files.

MEX-Plus has a facility, via the INSTALL command, to load optional extra code modules. One such module (the REMOTE module) allows MEX-Plus to be operated as a remote access system. I have used this with both direct wire and modem connections between machines. The former is handy when two machines are connected by a cable, as it relieves one of the need to run back and forth between the two keyboards to issue file transfer commands. I sometimes leave my system at work in remote mode so that I can call it from home to pick up a file that I forgot to take with me. Other optional modules support emulation of various terminals, including the VT100.

The TERM and TERMA commands open a file to record the incoming character stream from the modem. TERM creates a new file, while TERMA appends the new text to an existing file. The terminal mode commands T, L, and E described below can also take an optional 'A' suffix and capture file name. In those cases, once the capture file has been opened, MEX is put into terminal mode. The TERM and TERMA commands leave MEX in command mode for further script processing. The VIEW command allows the contents of the capture file to be reviewed while online, thereby affording some scroll-back capability (that is, a way to see text that has already scrolled off the screen). WRT closes the capture file (saves it) when you are done; DEL discards it.

The ALT command specifies an alternate drive/user area (in addition to the currently logged one) where MEX can search automatically for files (e.g., scripts, phone directory, etc.). The SEARCH command tells MEX how to go about searching for files, such as whether to search the alternate area before or after the logged area.

As you can probably guess already, it is not easy to remember all these commands and the syntax they require. That is where the HELP command comes in handy. It accesses an extensive HLP file (over 70K). There is also the wonderful CLONE command, which creates a new version of MEX (i.e., a new COM file) with the current configuration embedded—after the STAT, SET, TSET, and other commands have been used to change parameters to one's liking.

Telephone Interface Commands

MEX-Plus has two phone-dialing commands, CALL and DIAL. Both accept lists of either literal telephone numbers or names from the phone library. Numbers and names may be mixed. Numbers in the library may have associated baud rates, which will be selected automatically when that number is dialed. An optional parameter specifies the number of times to try connecting to the numbers before giving up, and the commands return a value to indicate which number, if any, was reached. Here is an example:

```
CALL LADERA 617-965-7259 #3
```

This will alternately called the Ladera Z-Node (whose number is in the library) and my Z-Node up to three times before giving up. The commands differ in that CALL puts one in terminal mode after a successful connection, while DIAL leaves one in command mode for further script processing.

There is also a way to get out of a connection. DSC will tell the modem to drop the connection. It can be configured (using a STAT command, of course) to use either the DTR (data terminal ready) hardware control line or the Hayes AT hangup command.

Terminal Commands

There are three commands for entering terminal mode. The command T (terminal) sets up a full-duplex terminal mode. Characters typed at the keyboard are sent to the modem; characters received from the modem are sent to the screen. If you are to see what you are typing, the remote system must echo back the characters it receives from you. Most microcomputer BBS systems do that.

The L (local echo) command sets up a half-duplex terminal mode. The difference is that the characters that you type are not only sent to the modem for transmission to the remote system; they are also echoed locally. This mode would be used with systems like GEnie that do not normally echo the characters they receive from you.

Finally, the E (echo) command sets you up as if you were a remote host. Every character you receive is then echoed back to the modem. If two people running MEX call each other (or two machines are hooked up by cable as I mentioned earlier), either both should be in L mode, or one should be in T mode and the other in E mode. I prefer the latter, since seeing the characters on the screen of the machine in T mode assures that the connection is working. I leave it as an exercise to the reader to figure out what will happen if both machines are in E mode. (Hint: feel free to experiment; you can get out of the infinite loop by exiting from terminal mode; you don't have to reboot.)

File Transfer Commands

Files are sent to the other computer using the S command, and received from the other computer using the R command. MEX-Plus supports three file transfer protocols: KERMIT, XMODEM, and YMODEM. The default protocol is set with the PROTO command. The protocol can also be specified explicitly using a prefix (K, M, or Y) to the S or R command.

The commands will also accept any of several suffixes as well. The suffix B indicates a batch-mode transfer. The K suffix with the S command indicates that the file should be sent in blocks of 1K bytes instead of the standard 128 bytes. If you append T, L, or E, you will return to the corresponding terminal mode after the transfer is completed. The D and X suffixes will tell MEX to disconnect from the remote system after the transfer is finished; with D you will return to MEX, while with X you will exit MEX as well. Here is an example.

```
YSBKX FILE1 FILE2 FILE3
```

This will send the three files using YMODEM batch with 1K blocks, hang up the phone, and exit from MEX.

Transactions with a host in the KERMIT server mode are supported with a number of special commands (KGET, KPUT, KBYE, KLOG, KFIN).

Video Commands

These are the commands that Rick Charnes loves so! There are commands for cursor addressing (@), beginning and ending the use of up to four video attributes (START and END), displaying special line-drawing graphics characters (DRAW), and creating lines and boxes (HLINE, VLINE, BOX). The screen can be cleared (CLS). It can also be turned on and off (SCREEN) so you can control what output is seen and what is not.

Variables

It is hard to do very sophisticated processing without variables. MEX-Plus offers variables of two types: numerical and string. The former are 16-bit integers (i.e., numbers from 0 to 65535); the latter are strings of up to 32 characters.

There are 26 numerical user variables designated by a percent sign followed by a letter (case does not matter, e.g., %a or %S). There are six string variables designated by the letters A through F. I've never run out of numerical variables, but I sure wish there were more string variables.

There are also two special numerical variables called VALUE and STACK. The former is used to hold the value returned from a number of MEX commands (DIR and DIAL/CALL were mentioned earlier). STACK is a more long-lived variable that can be exchanged in various ways with VALUE (the commands PUSH, POP, and XCHG). I think these are left over from earlier versions of MEX that did not offer the 26 user variables. There is little reason to use STACK any more. There are three special operations (ADD, SUB, and XOR) that can be performed on the VALUE variable. I've never found any use for these, either.

MEX can evaluate arithmetic expressions consisting of combinations of literal numbers, numerical variables, and arithmetic operators (+-*/). Here is an example that returns the least significant byte of the two-byte variable %V:

```
%V - 256 * ( %V / 256 )
```

Numbers can be entered in hexadecimal format by prefixing the number with a dollar sign ($100 is 256).

The value of an expression is assigned to a variable by the equal operator (=) as in %B=%A+3. NOTE: in most places in MEX-Plus, extra spaces may be included in command expressions. However, there are unfortunately a number of bugs in MEX-Plus, and some raise their heads in this area. Therefore, I recommend that extra spaces be omitted in working scripts (as opposed to fully commented reference versions, where extra spaces might be included to improve readability).

SAGE MICROSYSTEMS EAST

Selling & Supporting the Best in 8-Bit Software

- Automatic, Dynamic, Universal Z-Systems
  - Z3PLUS: Z-System for CP/M-Plus computers ($70)
  - NZCOM: Z-System for CP/M-2.2 computers ($70)
  - ZCPR34 Source Code: if you need to customize ($50)
- Plu*Perfect Systems
  - Backgrounder ii: CP/M-2.2 multitasker ($75)
  - ZDOS: date-stamping DOS ($75, $60 for ZRDOS owners)
  - DosDisk: MS-DOS disk-format emulator, supports subdirectories and date stamps ($30 - $45 depending on version)
- BDS C — Including Special Z-System Version ($90)
- Turbo Pascal — with New Loose-Leaf Manual ($60)
- SLR Systems (The Ultimate Assembly Language Tools)
  - Z80 Assemblers using Zilog (Z80ASM), Hitachi (SLR180), or Intel (SLRMAC) Mnemonics
  - Linker: SLRNK
  - TPA-Based ($50 each) or Virtual-Memory (Special: $160 each)
- ZMAC — Al Hawley's Z-System Macro Assembler with Linker ($50)
- NightOwl (Advanced Telecommunications)
  - MEX-Plus: automated modem operation with scripts ($60)
  - MEX-Pack: remote operation, terminal emulation ($100)

Next-day shipping of most products with modem download and support available. Order by phone, mail, or modem. Shipping and handling $3 per order (USA). Check, VISA, or MasterCard. Specify exact disk format.

Sage Microsystems East
1435 Centre St., Newton Centre, MA 02159-2469
Voice: 617-965-3552 (9:00am - 11:30pm)
Modem: 617-965-7259 (pw=DDT) (MABOS on PC-Pursuit)

Literal string expressions are composed by surrounding text with double-quote characters ("). The MEX manual says that values are assigned to string variables using the STORE command as in:

```
STORE "this is a test" TO A.
```

Experiment shows that the following simpler, undocumented syntax also works: A="test". Here I know from bitter experience that there should be no extra spaces around the equal sign. It often works, but not always. Most commands that accept quoted literal strings will also accept string variables (e.g., B=A or COMP A "yes").

Command line parameters are also available to scripts; they are passed in the form of variables represented by the numbers 1 through 9 surrounded by curly braces. These variables can always act as strings. If they express a number, they can also be used as numerical expressions. Thus, we might have A="{1}" or %A={1}. The latter expression will produce an error if the first command line token does not represent a number.

If the script invocation command line does not have a token referenced by an expression of the form {1}, {2}, etc., the script will bomb with an error message. The expression {n:default} al-

lows a default value to be used for parameter 'n' if none is given on the command line. This default value can be null, as in {1:}.

String variables can, like the command line tokens, be represented in expressions by curly braces around the letter (e.g., {B}). In such a case, the value of the expression is the string of characters alone, and double quote characters must surround the expression in most situations. There are exceptions. The SAY and SENDOUT commands (described in the next section) can be used directly with a variable, as in SAY A. The following two commands are equivalent:

```
SAY "Variable A has the value: ",A,"/n"
SAY "Variable A has the value: {A}/n"
```

The curly-brace expressions can be used to concatenate text, as in

```
A="{B} and {C}"
```

Expressions of this type are not documented and work only in script files; they will not work if entered directly at the MEX command prompt. Similar expressions can also be used to finesse variables into commands that normally do not take them. Here are some examples:

```
set baud {b}        (where, say, B="1200")
set baud {c}00      (where, say, C="24")
goto {1}            (where, say, token 1 is
"LOOP")
{a}                 (where, say, A="set baud
1200")
```

I have not figured out how to split a string variable into parts (such as words). I also had never been able to figure out a way to convert a numerical variable into a string. I still can't do it directly (things like B={%B} do not work), but the script in Figure 1 does it indirectly (inventions like this are part of the fun of writing this column). You may not be able to fully understand that script until you have read through the rest of the command descriptions.

Input/Output Commands

Programs generally are not terribly useful if there is no way to get data in or out. Here is what MEX-Plus offers.

The SAY command allows one to send characters to the screen. It accepts arguments of literal strings, string variables, and numerical expressions, as in

```
SAY "The sum is",%A+%B,"/n"
```

There are special character codes, such as "/n" (newline) or "/r" (return). Combined with the video commands mentioned earlier, the SAY command can produce some pretty fancy displays.

There is also the undocumented PRINT command that does almost exactly what SAY does, except that it does not need quotation marks around the literal text and does not interpret any special expressions. To PRINT, everything is a pure string. There are, thus, a couple of things PRINT can do that SAY cannot. Here are some examples:

```
PRINT Please enter "Hello" at the prompt.
PRINT Use the expression {1} for a token.
```

In the first case, PRINT allows one to send a double quote character to the screen. In the second case, a string that would be a variable expression can be displayed.

```
B=""                    initialize to null string
%y=%b                   set up scratch variable %y

LABEL LOOP              loop
%x=%y-10*(%y/10)        get the lowest digit ( %y MOD 10 )
if %x=0 B="0{B}"        preconcatenate the appropriate digit
if %x=1 B="1{B}"
if %x=2 B="2{B}"
if %x=3 B="3{B}"
if %x=4 B="4{B}"
if %x=5 B="5{B}"
if %x=6 B="6{B}"
if %x=7 B="7{B}"
if %x=8 B="8{B}"
if %x=9 B="9{B}"
%y=%y/10                divide number by 10
if %y>0 GOTO LOOP       continue until number is reduced to 0
```

Figure 1. Commented listing of a script that will convert the numerical variable %B into string form in string variable B, from which it can be used in various commands, such as: SET BAUD {B}. This code could be made into a subroutine by adding the command PROC NUM2STR at the beginning and ENDP at the end. If you try this script, do not enter the comments, of course.

The SENDOUT command is used to send literal text or the contents of string variables to the modem. The PREFIX and SUFFIX commands can be used to define strings that are automatically sent before and after the designated text to save one the trouble of having to include certain characters (such as, perhaps, a carriage return and linefeed) explicitly with each string. Longer, fixed strings can be sent using the TRANSMIT command, which sends the contents of a file just as if you were typing it in terminal mode. A pair of delay constants controls the speed with which this "automatic typist" performs.

The INPUT command allows interactive entry of the value for a numerical variable; ACCEPT does the same thing for a string variable. The TIME and DATE commands allow one to access a real-time clock. Besides displaying the information on the screen, the commands are supposed to put the corresponding data into the VALUE variable. There is a bug here, and the time value is used by both commands. There appears to be no way to determine the date from within a script. The PEEK and POKE commands allow one to look at and modify memory for the ultimate in hacking from a script! (That probably means that there would be a way to find the date if one really needed it badly. It also means that the complete Z-System environment can be accessed.)

Flow Control Commands

Like Z-System, MEX-Plus has flow control commands to allow a script to perform tests and to act differently depending on the results. There is the standard set of flow commands IF, ELSE, and ENDIF. They support 8 levels of nesting (just like Z-System). Here is an example:

```
IF %B=1200
  B="1200"
ELSE
  B="2400"
ENDIF
```

Additionally, there is a single-line IF command. It is distinguished from the multiple-line IF by a command (a 'then' clause) that is part of the IF statement. Here is an example:

```
IF %B=2400 B="2400";SAY "2400 bps";GOTO CONTINUE
```

With the single-line IF, when the test fails, the entire line is ignored. Please note that had there been a semicolon after the "IF %B=2400", then this would have been a multiple-line IF (albeit with several of its

'lines' on one line).

The single-line IF command comes in especially handy, because it is generally awkward to perform a GOTO jump out of an IF/ELSE/ENDIF block. Although the manual warns against it, there is (I'm pretty sure) no harm per se in doing it. It is just that you have to make sure that the ENDIF is not skipped lest you get nested deeper and deeper. As with Z-System, you have to make sure the IFs get terminated; unfortunately, this is not so easy, because, unlike Z-System, MEX-Plus has no XIF or ZIF command. Here is a very convoluted example of a way in which it could be done:

```
if %b
  say "TRUE/n"
  goto cont
endif
.
say "FALSE/n"
goto done
.
label cont
endif
.
label done
say ''Now we are done/n''
```

There is also a SKIPIF command. If its test is true, then the next command is skipped. Note well that while the IF command may skip an entire line of commands, SKIPIF skips only one command, no matter how many commands may appear on the same line.

Numerical Tests

Flow control would not be very useful without ways to test things. First we will consider tests on numbers.

Test results are expressed numerically, with 0 representing false and non-zero (usually 1) representing true. You can see this for yourself by entering the command "SAY 1<2" or "SAY 1>2". The following logical operators can be used for comparisons: equals (=), not equals (<>), less than (<), less than or equal to (<= or =<), greater than (>), or greater than or equal to (>= or =>). Note that the MEX manual has a misprint in one place and gives the not-equal-to operator as '!'. That is incorrect and will not work.

Although comparison tests return a numerical value, those values for some reason cannot be used in arithmetic expressions. In other words, you can't have (%A>%B)*(%A<%C). As far as I can tell, this means that you cannot perform compound tests, such as "IF %A>%B AND %A<%C". Performing such compound tests is made more difficult by the fact that the 'then' clause of a single-line IF cannot be another IF. If you won't be using GOTO, nested multi-line IF commands will do the trick. Otherwise, you might have to resort to some explicit arithmetic as in the following:

```
%z=1
if %a<=%b then %z=0
if %a>=%c then %z=0
if %z then ....
```

We started out assuming TRUE (%z=1). Then if either condition was false, we set %z to false.

Before we leave this topic (I know we've been here a long time), I have to mention that MEX has a bug that causes it to issue fallacious but very annoying error reports when comparison operations are performed with numerical variables having particular values. I have carried out a number of experiments to try to determine the exact circumstances under which this problem occurs, but so far I cannot fathom a pattern. The trouble often appears, however, with comparisons to standard data rate values, such as 300. To get around the problem, I sometimes divide the variable by 100, compare it to 3, 12, or 24, and then multiply it by 100 to restore its original value. What a pain!

String Tests

Strings are compared using the COMP command. It accepts two strings, each of which can be either a literal string or a string variable. The result of the comparison is returned in the VALUE variable, which can then be tested for a value of 0 or 1. The STAT CASE setting determines whether the comparison will be case sensitive or not. There is a bug with the COMP command; it will not give the answer 1 (true) when both strings are null (""), though it will work if only one string is null. If you want to see if a command line token was given, you can use the following tests:

```
COMP "{1:} " " "
COMP "{1:}X" "X"
```

An extra character (space or 'X') is concatenated to the string represented by {1:}. You might also use

```
COMP "{1:null}" "null"
```

Here, the first parameter cannot be null. If token 1 is not given, the default value "null" will be used instead. Of course, if the user enters "null", the same result will be obtained. The two earlier examples are, thus, more robust.

There is one oddball command that I don't know where to put: SLEEP. It is sort of a flow control command, so I'll stick it here. It just tells the system to go to sleep (pause) for a designated time interval.

Program Control Commands

MEX-Plus supports several script program structures. The main unit of a script program is a script file, which has a file extension of MEX. It can be invoked as a main program by the READ command. It can also be invoked as a subprogram by the DO command, which allows scripts to be nested. I do not know how deep this nesting can be, but I just tested it to five levels. The STOP command is used to exit from a READ or DO command. In the former case, control returns to the MEX program (possibly in terminal mode); in the latter, control returns to the script that called the current script. A script also terminates automatically at the end-of-file. The STOP command unfortunately displays an annoying message about the script being aborted. To exit gracefully, it is better to put a label at the end (e.g., LABEL END) and to exit using GOTO END.

A READ command may be given inside a script. In this case, control is transferred to the new script, which overwrites the old script in memory. With the DO command, the new script lines are read into memory along with the currently running script.

I always enjoy writing these columns because I end up asking some new questions and learning some new answers. Just now, to see how MEX works, I was examining the memory image after a MEX script file ran. First, I learned that the script text is stored backwards in memory starting from near the top. I verified that after each DO is finished, the memory is reclaimed and is available for use by another subroutine script. I also discovered that the entire script file, including all comments, is read into memory.

We can draw some important conclusions from these observations about how complex scripts should be implemented. First, there are several advantages to using versions of script files from which comments have been stripped. The files will, of course, load

faster, and there will also be more room in memory for such things as file transfer buffers. Many MEX users have run into problems of insufficient memory while running complex scripts. Second, it is a good idea to chain from one script to another rather than building everything into a single script. In my PC-Pursuit script, I chain to a very small script just after the remote system has been reached and the script is about to put me into terminal mode.

Structure is permitted within an individual script file as well in the form of internal subroutines. Subroutines begin with a PROC (procedure) command and end with an ENDP (end-procedure) command. They are invoked by the GOSUB command. Again, I do not know how deeply they may be nested, but I just tested them to 9 levels.

Script files may also contain unstructured program groupings (that anathema to modern structured programming). The LABEL command allows any point in the script to be given a name, and the GOTO command allows a branch to that point. The manual indicates that the name may have up to 16 alphabetic characters and warns that the line with the LABEL command may not have any other commands on the line. I know that I have run afoul of that restriction in the past, but, oddly enough, in my testing now I was totally unable to generate a problem. I tried everything I could think of: an immediate semicolon, a semicolon after a space or a tab, tabs after the semicolon. They all worked just fine (how could MEX tell that this was just a test?).

The manual is clearly wrong when it says that only alphabetic characters can be used. I assumed that it really meant alphanumeric, but in my experiments I learned that absolutely any characters can be used, including spaces! Here are the rules that emerged from my testing. First, all spaces and tabs after the LABEL command are ignored. That point marks the beginning of the name. Next one starts from the end of the line or the command separator character and strips all tabs and spaces backward. That point marks the end of the name. What is between those marks is taken as the label. For example, one can have a label of "ENTRY 1" (with the embedded space and with or without the quote characters, in fact). Mind you, I am not by any means suggesting that you use such labels. As I mentioned before, I have not always found MEX to behave exactly the way it did in these experiments. If anyone can figure this out more precisely, I would really like to hear about it.

The other thing I learned from these experiments is that MEX always scans for a label from the beginning of the script. This means that if you use the same label a second time, the second occurrence will never be found.

One Script Example

I just don't feel right about presenting all this information about MEX-Plus commands without showing at least one real-life example. Figure 2 shows the final script in my PC-Pursuit script suite. Once I have successfully connected to the outdial city and reached the remote system there, I chain to this script. Let's look at it line by line.

The first line begins with a period, so it is a comment. I always include a title line and often add some description of the function of the script and the parameters it takes.

The real work of the script begins at line 3. In many parts of the script, I do not want the output from commands to show on the screen. Now I do, so I issue the "SCREEN ON" command. I'd also like to start with a clean screen, so I issue CLS, too.

```
01  .. PCPCONN SCRIPT -- Connected to Destination System
02
03  screen on
04  cls
05  say "Connected to ",F," at ",B,"00 bps/n/n"
06  t
07
08  LABEL LOOP
09  say "/nEnter single MEX command (or M for menu): "
10  accept A
11  comp A "M"
12  if value=1 READ PCPMENU
13  {A}
14  GOTO LOOP
```

Figure 2. This is the final script in my PC-Pursuit suite. Just after connecting to a remote system, I chain to this script to free up as much memory as possible for other uses, such as capture buffers and/or file transfer buffers. Line numbers have been added for reference purposes in the text.

In line 5, the script tells the user the name of the system that has been reached (that was previously stored in string variable F) and at what baud rate (previously stored in string variable B). Then the script drops one into terminal mode, where one can work interactively as long as one likes.

When terminal mode is exited (by pressing <ESC> E), the script resumes at the label LOOP. Line 9 prompts the user to enter a command. If the user enters "M", then the main menu script is run, allowing one to call another system in another or the same city. Line 10 accepts the answer from the user and places it into string variable A.

In line 11, the user's answer is compared to the string "M". If it was "M", then the variable VALUE will have the value 1 (true), and line 12 will cause the script to chain to the script file PCPMENU.MEX. If the user entered anything other than "M", then execution will continue at line 13, where the user's command is executed. When that command has completed, control will return to line 14, which branches back to label LOOP and a prompt for another command.

The user's command can be just about anything. For example, if it is "T", then MEX will enter terminal mode for more interactive work. The one restriction I have found, is that only a single command can be entered. A multiple command line, with commands separated by semicolons (or whatever the designated SEP character is), does not work for reasons I do not yet understand. Maybe I will have figured it out (or one of you will have) by next time, when I will cover the real guts of the PC-Pursuit script. •

MOVING?

Make certain that TCJ follows you to your new address. Send both old and new address along with your expiration number that appears on your mailing label to:

THE COMPUTER JOURNAL
190 Sullivan Crossroad
Columbia Falls, MT 59912

If you move and don't notify us, TCJ is not responsible for copies you miss. Please allow six weeks notice. Thanks.

THE COMPUTER JOURNAL

Back Issues

Special Close Out Sale
on these back issues only
Issues—1, 2, 3, 4 and 8
3 or more, $1.50 each postpaid in the U.S.
Outside of the U.S., 3 or more, $2.50 each postpaid surface.
Other back issues are available at the regular price.

Issue Number 1:

- RS-232 Interface Part One
- Telecomputing with the Apple II
- Beginner's Column: Getting Started
- Build an ""Epram"

Issue Number 2:

- File Transfer Programs for CP/M
- RS-232 Interface Part Two
- Build Hardware Print Spooler: Part 1
- Review of Floppy Disk Formats
- Sending Morse Code with an Apple II
- Beginner's Column: Basic Concepts and Formulas

Issue Number 3:

- Add an 8087 Math Chip to Your Dual Processor Board
- Build an A/D Converter for Apple II
- Modems for Micros
- The CP/M Operating System
- Build Hardware Print Spooler: Part 2

Issue Number 4:

- Optronics, Part 1: Detecting, Generating, and Using Light in Electronics
- Multi-User: An Introduction
- Making the CP/M User Function More Useful
- Build Hardware Print Spooler: Part 3
- Beginner's Column: Power Supply Design

Issue Number 8:

- Build VIC-20 EPROM Programmer.
- Multi-User: CP/Net.
- Build High Resolution S-100 Graphics Board: Part 3.
- System Integration, Part 3: CP/M 3.0.
- Linear Optimization with Micros.

Issue Number 16:

- Debugging 8087 Code
- Using the Apple Game Port
- BASE: Part Four
- Using the S-100 Bus and the 68008 CPU
- Interfacing Tips & Troubles: Build a ""Jellybean" Logic-to-RS232 Converter

Issue Number 18:

- Parallel Interface for Apple II Game Port
- The Hacker's MAC: A Letter from Lee Felsenstein
- S-100 Graphics Screen Dump
- The LS-100 Disk Simulator Kit
- BASE: Part Six
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 1

Issue Number 19:

- Using The Extensibility of Forth
- Extended CBIOS
- A $500 Superbrain Computer
- BASE: Part Seven
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 2
- Multitasking and Windows with CP/M: A Review of MTBASIC

Issue Number 20:

- Designing an 8035 SBC
- Using Apple Graphics from CP/M: Turbo Pascal Controls Apple Graphics
- Soldering and Other Strange Tales
- Build a S-100 Floppy Disk Controller: WD2797 Controller for CP/M 68K

Issue Number 21:

- Extending Turbo Pascal: Customize with Procedures and Functions
- Unsoldering: The Arcane Art
- Analog Data Acquisition and Control: Connecting Your Computer to the Real World
- Programming the 8035 SBC

Issue Number 22:

- NEW-DOS: Write Your Own Operating System
- Variability in the BDS C Standard Library
- The SCSI Interface: Introductory Column
- Using Turbo Pascal ISAM Files
- The AMPRO Little Board Column

Issue Number 23:

- C Column: Flow Control & Program Structure
- The Z Column: Getting Started with Directories & User Areas
- The SCSI Interface: Introduction to SCSI
- NEW-DOS: The Console Command Processor
- Editing The CP/M Operating System
- INDEXER: Turbo Pascal Program to Create Index
- The AMPRO Little Board Column

Issue Number 24:

- Selecting and Building a System
- The SCSI Interface: SCSI Command Protocol
- Introduction to Assembly Code for CP/M
- The C Column: Software Text Filters
- AMPRO 186 Column: Installing MS-DOS Software
- The Z Column
- NEW-DOS: The CCP Internal Commands
- ZTIME-1: A Realtime Clock for the AMPRO Z-80 Little Board

Issue Number 25:

- Repairing & Modifying Printed Circuits
- Z-Com vs Hacker Version of Z-System
- Exploring Single Linked Lists in C
- Adding Serial Port to Ampro L.B.
- Building a SCSI Adapter
- New-Dos: CCP Internal Commands
- Ampro '186 Networking with SuperDUO
- ZSIG Column

Issue Number 26:

- Bus Systems: Selecting a System Bus
- Using the SB180 Real Time Clock
- The SCSI Interface: Software for the SCSI Adapter
- Inside AMPRO Computers
- NEW-DOS: The CCP Commands Continued
- ZSIG Corner
- Affordable C Compilers
- Concurrent Multitasking: A Review of DoubleDOS

Issue Number 27:

- 68000 TinyGiant: Hawthorne's Low Cost 16-bit SBC and Operating System
- The Art of Source Code Generation: Disassembling Z-80 Software
- Feedback Control System Analysis: Using Root Locus Analysis and Feedback Loop Compensation
- The C Column: A Graphics Primitive Package
- The Hitachi HD64180: New Life for 8-bit Systems
- ZSIG Corner: Command Line Generators and Aliases
- A Tutor Program for Forth: Writing a Forth Tutor in Forth
- Disk Parameters: Modifying The CP/M Disk Parameter Block for Foreign Disk Formats

Issue Number 28:

- Starting your Own BBS
- Build an A/D Converter for the Ampro L.B.• HD64180: Setting the wait states & RAM refresh, using PRT & DMA
- Using SCSI for Real Time Control
- Open Letter to STD-Bus Manufacturers
- Patching Turbo Pascal
- Choosing a Language for Machine Control

Issue Number 29:

- Better Software Filter Design
- MDISK: Adding a 1 Meg RAM disk to Ampro L.B., part one.
- Using the Hitachi HD64180: Embedded processor design.
- 68000: Why use a new OS and the 68000?
- Detecting the 8087 Math Chip
- Floppy Disk Track Structure
- The ZCPR3 Corner

Issue Number 30:

- Double Density Floppy Controller
- ZCPR3 IOP for the Ampro L.B.
- 3200 Hacker's Language
- MDISK: 1 Meg RAM disk for Ampro LB, part 2
- Non-Preemptive Multitasking
- Software Timers for the 68000
- Lilliput Z-Node
- The ZCPR3 Corner
- The CP/M Corner

Issue Number 31:

- Using SCSI for Generalized I/O
- Communicating with Floppy Disks: Disk parameters and their variations.
- XBIOS: A replacement BIOS for the SB180.
- K-OS ONE and the SAGE: Demystifing Operating Systems.
- Remote: Designing a remote system program.
- The ZCPR3 Corner: ARUNZ documentation.

Issue Number 32:

- Language Development: Automatic generation of parsers for interactive systems.
- Designing Operating Systems: A ROM based O.S. for the Z81.
- Advanced CP/M: Boosting Performance.
- Systematic Elimination of MS-DOS Files: Part 1, Deleting root directories & an in-depth look at the FCB.
- WordStar 4.0 on Generic MS-DOS Systems: Patching for ASCII terminal based systems.
- K-OS ONE and the SAGE: Part 2, System layout and hardware configuration.
- The ZCPR3 Corner: NZCOM and ZCPR34.

Issue Number 33:

- Data File Conversion: Writing a filter to convert foreign file formats.
- Advanced CP/M: ZCPR3PLUS, and how to write self relocating Z80 code.
- DataBase: The first in a series on data bases and information processing.
- SCSI for the S-100 Bus: Another example of SCSI's versatility.
- A Mouse on any Hardware: Implementing the mouse on a Z80 system.
- Systematic Elimination of MS-DOS Files: Part 2—Subdirectories and extnded DOS services.
- ZCPR3 Corner: ARUNZ, Shells, and patching WordStar 4.0

Issue Number 34:

- Developing a File Encryption System.
- Database: A continuation of the data base primer series.
- A Simple Multitasking Executive: Designing an embedded controller multitasking executive.
- ZCPR3: Relocatable code, PRL files, ZCPR34, and Type 4 programs.
- New Microcontrollers Have Smarts: Chips with BASIC or Forth in ROM are easy to program.
- Advanced CP/M: Operating system extensions to BDOS and BIOS, RSXs for CP/M 2.2.
- Macintosh Data File Conversion in Turbo Pascal.
- The Computer Corner

Issue Number 35:

- All This & Modula-2: A Pascal-like alternative with scope and parameter passing.
- A Short Course in Source Code Generation: Disassembling 8088 software to produce modifiable assem. source code.
- Real Computing: The NS32032.
- S-100: EPROM Burner project for S-100 hardware hackers.
- Advanced CP/M: An up-to-date DOS, plus details on file structure and formats.
- REL-Style Assembly Language for CP/M and Z-System. Part 1: Selecting your assembler, linker and debugger.
- The Computer Corner

Issue Number 36:

- Information Engineering: Introduction.
- Modula-2: A list of reference books.
- Temperature Measurement & Control: Agricultural computer application.
- ZCPR3 Corner: Z-Nodes, Z-Plan, Amstrand computer, and ZFILE.
- Real Computing: NS32032 hardware for experimenter, CPUs in series, software options.
- SPRINT: A review.
- REL-Style Assembly Language for CP/M & ZSystems, part 2.
- Advanced CP/M: Environmental programming.
- The Computer Corner.

Issue Number 37:

- C Pointers, Arrays & Structures Made Easier: Part 1, Pointers.
- ZCPR3 Corner: Z-Nodes, patching for NZCOM, ZFILER.
- Information Engineering: Basic Concepts: fields, field definition, client worksheets.
- Shells: Using ZCPR3 named shell variables to store date variables.
- Resident Programs: A detailed look at TSRs & how they can lead to chaos.
- Advanced CP/M: Raw and cooked console I/O.
- Real Computing: The NS 32000.
- ZSDOS: Anatomy of an Operating System: Part 1.
- The Computer Corner.

Issue Number 38:

- C Math: Handling Dollars and Cents With C.
- Advanced CP/M: Batch Processing and a New ZEX.
- C Pointers, Arrays & Structures Made Easier: Part 2, Arrays.
- The Z-System Corner: Shells and ZEX, new Z-Node Central, system security under Z-Systems.
- Information Engineering: The portable Information Age.
- Computer Aided Publishing: Introduction to publishing and Desk Top Publishing.
- Shells: ZEX and hard disk backups.
- Real Computing: The National Semiconductor NS320XX.
- ZSDOS: Anatomy of an Operating System, Part 2.

Issue Number 39:

- Programming for Performance: Assembly Language techniques.
- Computer Aided Publishing: The Hewlett Packard LaserJet.
- The Z-System Corner: System enhancements with NZCOM.
- Generating LaserJet Fonts: A review of Digi-Fonts.
- Advanced CP/M: Making old programs Z-System aware.
- C Pointers, Arrays & Structures Made Easier: Part 3: Structures.
- Shells: Using ARUNZ alias with ZCAL.
- Real Computing: The National Semiconductor NS320XX.
- The Computer Corner.

Issue Number 40:

- Programming the LaserJet: Using the escape codes.
- Beginning Forth Column: Introduction.
- Advanced Forth Column: Variant Records and Modules.
- LINKPRL: Generating the bit maps for PRL files from a REL file.
- WordTech's dBXL: Writing your own custom designed business program.
- Advanced CP/M: ZEX 5.0—The machine and the language.
- Programming for Performance: Assembly language techniques.
- Programming Input/Output With C: Keyboard and screen functions.
- The Z-System Corner: Remote access systems and BDS C.
- Real Computing: The NS320XX
- The Computer Corner.

Issue Number 41:

- Forth Column: ADTs, Object Oriented Concepts.
- Improving the Ampro LB: Overcoming the 88Mb hard drive limit.
- How to add Data Structures in Forth
- Advanced CP/M: CP/M is hacker's haven, and Z-System Command Scheduler.
- The Z-System Corner: Extended Multiple Command Line, and aliases.
- Programming disk and printer functions with C.
- LINKPRL: Making RSXes easy.
- SCOPY: Copying a series of unrelated files.
- The Computer Corner.

Issue Number 42:

- Dynamic Memory Allocation: Allocating memory at runtime with examples in Forth.
- Using BYE with NZCOM.
- C and the MS-DOS Screen Character Attributes.
- Forth Column: Lists and object oriented Forth.
- The Z-System Corner: Genie, BDS Z and Z-System Fundamentals.
- 68705 Embedded Controller Application: An example of a single-chip microcontroller application.
- Advanced CP/M: PluPerfect Writer and using BDS C with REL files.
- Real Computing: The NS 32000.
- The Computer Corner

Issue Number 43:

- Standardize Your Floppy Disk Drives.
- A New History Shell for ZSystem.
- Heath's HDOS, Then and Now.
- The ZSystem Corner: Software update service, and customizing NZCOM.
- Graphics Programming With C: Graphics routines for the IBM PC, and the Turbo C graphics library.
- Lazy Evaluation: End the evaluation as soon as the result is known.
- S-100: There's still life in the old bus.
- Advanced CP/M: Passing parameters, and complex error recovery.
- Real Computing: The NS32000.
- The Computer Corner.

Help TCJ Grow

TCJ is expanding, and we need to inform others about what we are doing.

You can help TCJ grow by distributing flyers at meetings, swap meets, etc. Tell us the date of the event and how many flyers you need, and we'll provide them. Allow three to four weeks to print and ship the flyers.

Increasing the number of subscribers will enable TCJ to provide more pages of vital information—your efforts in finding more subscribers will be appreciated by all.

Tell Your Friends

Subscriptions	**U.S.**	**Foreign (Surface)**	**Total**
1year (6 issues)	$18.00	$24.00	
2 years (12 issues)	$32.00	$46.00	
Air Mail rates on request.			
Back Issues			
16 thru #43	$3.50 ea.	$4.50 ea.	
6 or more	$3.00 ea.	$4.00 ea.	
#44 and up	$4.50 ea.	$5.50 ea.	
6 or more	$$.00 ea.	$5.00 ea.	

Issue #s ordered_____

Subscription Total _____

Back Issues Total _____

Total Enclosed _____

All funds must be in U.S. dollars on a U.S. bank

Name_____

Address_____

_____

_____

☐ Check ☐ VISA ☐ MasterCard Exp. Date____

Card #_____

Signature_____

The Computer Journal
190 Sullivan Crossroad, Columbia Falls, MT 59912
Phone (406) 257-9119 Mountain Time Zone

Xerox 16 / 8 DEM-II Computers

New dual system computers with the Disk Expansion Module. These systems include the following:

- Z80A 4 MHz CPU with 64 K of RAM
- 8086 4.77 MHz CPU with 128 K RAM
- 2 Serial ports
- 1 Parallel port
- 10 Meg 5.25" hard drive (**NOT** 8")
- 322 K DSDD floppy drive
- Low-profile programmable keyboard
- Monitor

CP/M-80 2.2, CP/M-86, and "Select" word processor are included. MS-DOS 2.01 is available as an option for an additional $35.

Cost is $329 plus $50 shipping in the US. This also includes a one year subscription to The Computer Journal (current subscribers should include a photocopy of their label so that their subscription can be extended). Registered owners of NZCOM receive a discount of $15. If you order NZCOM **at the time of the order,** deduct the $15. Order by personal check, bank cashier's check or money order. Personal checks held ten days. Allow 4 to 6 weeks for delivery.

Chris McEwen – Socrates Z Node 32
PO Box 12, S. Plainfield, NJ 07080
(201) 754-9067 3/12/24 bps

Technology Resources

K-OS ONE—Single user generic 68000 operating system for your 68000 hardware. It uses the MS-DOS disk format, and includes the operating system with source code (written in HTPL), an editor, assembler, and HTPL compiler. A sample BIOS code and a boot loader are included. This is **not** ready-to-run—you have to install the BIOS on your system, but the source code and language compiler are included **$50**

HT-Forth—A full featured, interactive Forth that works with the K-OS ONE operating system. It uses a full 32 bit stack and 32 bit arithmetic to take full advantage of the 68000. Programs are position independent and are limited in size only by the memory available. Source code compiles to inline macros, JSR, or BSR so there is no inner interpreter overhead. Standard ASCII files are used. Includes full screen editor and a Forth style 68000 assembler $100

68000Cross Assembler—Written entirely in 8086 assembly language, it is small and fast. All input and output is done with standard MS-DOS calls so it will run on any MS-DOS system, even those which are not totally PC compatible. All 68000 and 68010 instructions are supported. It has conditional assembly, the symbol table is in alphabetical order, and cross referencing is included. Include files are supported so it is easy to assemble big programs, but edit them in small pieces. An equate file can be produced for PROM based programming $50

ORDER FROM

Technology Resources
190 Sullivan Crossroad
Columbia Falls, MT 59912
Phone (406) 257-9119

Visa and Mastercard accepted
Prices postpaid in the U.S. and Canada

Letters

I'd Like to See..

Upgrading non-IEEE 696 machines (i.e. Altair IMSAI, etc.) to current specs. This information would help on assembling systems from what cards are still available. Maybe the guys at Fulcrum could contribute to this. They may be the only maker of S-100 stuff left. I have done some work on this, but am still having problems making stuff work. I obviously haven't hit on all the changes needed.

Language articles along the theme of Modula-2 (or C, or ADA, etc.) as a second language, and gear it to those who know BASIC and a little assembler (i.e. BIOS hacking). Yes, the languages are very different conceptually, but still it might get some of us going.

Hints on CP/M dBASE II: What revs had what bugs, what is the last rev, what functions were in what overlays (there are 8-10 as I recall)—if you don't need certain functions you can leave some off the work disk, but which ones? As I recall, you had a data base series going. Why not design and write a dBASE file compatible system? DB II was the *only* widely used (and therefore sworn at) data base on CP/M, and there would be a big interest in a cheap replacement that did more.

How about an article on multi-tasking? I've written a small multi-task scheduler, and in fact have had CP/M 2.2 running as a task. I'm no systems programmer, but could present some basic theory.

Regarding Postscript language: how about an article on how to drop a company logo on a page? Or a signature? and how does Postscript differ from HP Graphics Language?

How about the differences in the various Mushy-DOS revs? Here at work we've got 2.11, 3.1, 3.2, and 3.3 on various machines. One set of machines (the original HP Vectra) **MUST** run 3.1, and nothing else. I've got a Vectra ES/12 on my desk running 3.21, and another guy here has 3.3 on his Vectra. What has he got that I haven't? (and what makes the stuff so revision sensitive?)

How about printer upgrades? I was given an Okidata u83. Does anybody still sell the "more-fonts" kits? How does an Epson 286 differ from a 286E? or an FX-100 from an FX-100 III?

One of my long term projects has been to rewrite my amateur radio repeater operating system and my floppy formatter. The repeater program is over 1,200 lines, and the formatter over 800 lines (and a friend has repeater code over 25,000 lines. He has to use a cross-assembler on a AT). Even under Z80ASM, on a ram disk and with a print spooler the formatter takes 15 minutes to assemble (even without generating a listing). There has to be a better way, and there is: relocatable modules. Modify a module, assemble it by itself, link it and fire up the debugger. But I have yet to find a good book on Z80 assembler programming that shows how to use the EXTRN, CSEG, DSEG (and similar family) statements. A few people have suggested looking at other programs and reading the comments, but nothing beats a **good** book. I thought the Microsoft have written one oriented at M80 / L80 / LIB80, but if they did I haven't found it. Maybe Steve at SLR could write a couple of articles on assembler software development techniques and tools.

How about an article on SCSI. The Commodore Amiga 1000 has one serial port. A 4-port serial card that hangs on the SCSI port, with a software driver for it would be VERY popular.

Maybe an overview on X.25 protocol, and how it's used. What's a PAD anyway, and how does it work, and what services does it provide.

How about an Ethernet overview? There are several different flavors of IEEE 802, some of which are compatible, some are not (i.e. can't be mixed on the same cable). And what's TCP/IP anyway?

I could go on and on, but then you'd never get this renewal check, and I'd never get back to work.

M.M.

Forth

Keep up the good work! I especially like your Forth columns and hope they will be permanent features of TCJ.

In your discussion of single chip micros with Forth kernals, I believe you skipped Zilog's "Super-Z8" chip. This chip has two machine instructions which implement ":" and ";" directly for direct threaded code. At one time Zilog had evaluation boards available. I believe that the Forth kernal / development system is available through Inner Access Corp. (PO Box 888, Belmont, CA 94002). The instruction set is simple and powerful.

P.H.

•

Editor

(Continued from page 2)

Technical Consulting

Consulting is a very attractive alternative for people who have been cast out in the current surge of mergers, downsizings, and reductions in staff—and for those who are tired of waiting for the ax to fall. The lack of long term job security has prevented many from entering the ranks of the consultants. Now that there is no longer any job security in working for a major company, many people are finding the consulting field very attractive. It's especially attractive to those who have been laid off and who have not been able to find another position, even after sending out hundreds of résumés.

Companies are using consultants to provide short term technical skills, rather than adding employees, and there is a growing need for high tech consultants. TCJ is researching this topic, and is planning a special issue on consulting. Submit your comments, suggestions, and article proposals.

Embedded Controllers

Our embedded controller projects are progressing well, but did not make it in this issue. Scheduled for #45 are the beginning of a tutorial on *Getting Started with Embedded Controllers* and a series on serial and parallel communications with the Z80. Many projects will be based on controller chips, such as the 8031, 68XX, and Z8, but the Z80 is still attractive for certain applications—especially now that a 20 MHz version is available.

There are dozens of microcontroller chips available, and it is tempting to spend all of the available time evaluating the various chips rather than to make a decision and to actually get started on the design. I know that I've fallen into that trap and spent too much time researching, but now I'm working on a project for the next issue.

I've decided to base the first project on the Z80 microprocessor instead of one of the various microcontrollers. Extensive discussion on selecting the chip will be part of the series. Briefly, my reasons for starting with the Z80 are that I have the CPU and peripheral chips, I have the assembler and debugger, I am familiar with the Z80, and serial and parallel I/O is a major function in this project. It is a good opportunity to become familiar with embedded controller functions and I/O. •

Reader Letters

The reader letters column died because of a lack of your letters. But it's too useful and interesting to lose. It's there to provide you a place to have a say in where we go—but we need your letters.

patibility. The manual indicates that all IF THEN statements must have the IF and THEN on the same line. Avocet did that, however in reality the Motorola assembler didn't care. I have had to go and edit all text files and put the THENs and DOs (of while..do) on one line. I guess following the book closely was not so good – maybe 90% compatible would have been better.

If my statement of IF THEN structure in an assembler made you ask, "what?", yup that is right, Motorola assembler has all kinds of structure in it. There is the standard conditional assembly structure, include this code if...(debug?). There also is structured assembly statements to allow for a higher level approach to handling conditional test. These are the IF..THEN...ELSE and WHILE...DO type of statements. The Avocet manual tells you what they actually become, while Motorola keeps it a secret. Typically they just turn into a compare this to that and jump here if not true, else do that. Which is what I have been doing for years and find easier to deal with. The programmers who wrote the 68000 code had little if any assembly experience and so they used the structures endlessly and senselessly. They cause almost as much problems as the MACROs.

The only thing the programmers loved more than structure was MACROS. There are macros for everything. Even things that could be done simpler and easier without them are done with macros. I spent one whole day just trying to figure out two simple macros. One was to help solve the need to put "0" in a table to pad out the lines to 14 characters. The system disassembler uses tables to step through and compare the code to, and then prints out the mnemonic. Well each of the mnemonics is different in length and needed padding to make them all the same length. The macro was blowing up by calling itself more than 30 times (a limit in Avocet). I solved the problem by putting in "0" as needed to pad out the lines, a considerable more simple solution I admit, but it works correctly – always!

Another macro problem popped up due to the Motorola assembler always changing lower case character to upper, even in the macro calls. Avocet does not. The macro did a character string check and was failing the string test because one was lower case the other upper. That was easy to solve (make them both the same – lower case), but if the Avocet had not included a macro preprocessor I would have spent more than one day on finding the problem. On a PC based system, memory is a problem, so the assembly process is broken into two steps. You process MACROs using the M68K macro preprocessor and it creates an A68 text file. The new file contains the MACROs with flags stating if the test were true or false (as well as other helpful comments). All that was needed to see where my MACRO was going wrong was to check the A68 file. It showed that string compares were turning out false not true as had been intended.

Overall I like the Avocet assembler and think it will work just fine. I will need to work on all 200 or more files, checking for IF THEN and bad MACROS, but an hour or so each should do it. I also discovered that the assembler was originally a Quello assembler. That is what I had been using at home, Quello version 1, I think. It was a public domain program way back when. They are now on version 6 and have linkers, library editors, and the macro preprocessor (which will be my life saver). All in all I think it is a good and sound product.

MCUs

We have been talking about using small computers to do things. I like the Motorola's brand of micros over the Intel's. At work they have used an 68705 as keyboard translator. We have our own style of keyboard, and many clients want to use PC compatible keyboards in their place. One of the engineers designed a three chip product (68705 and two glue TTLs) to convert the PC operation into one compatible to ours. When I got my February issue of Circuit Cellar INK (a Ciarcia publication) it too contains a keyboard translation design. It uses an 8031 (Intel) with external ROM. The main difference is the ability of this design to supply serial or parallel ASCII output. The article explains how to set it up for whatever operation you need. It seems several people have decided using small controllers is more fun and maybe cheaper than buying new keyboards.

As to why I would chose a Motorola over Intel product, it is their instruction set by far. It is simple, and linear. You may be wondering what linear means in this case. It is consistency. Whether you are talking to memory or I/O it is all addressed and handled the same. Internal registers typically are addressed and manipulated the same, no special instructions or special operations. Whenever I work on PCs, it seems I am always looking up some special condition or operation. The 80X86 set has lots of exception instructions, by that I mean you do it this way on these registers, but this other way for those. Motorola's instruction set is, "you do it this way for everything." The only device I have never minded the special instructions is the Z80. In the case of the Z80, I have two reasons; I have spent a large amount of time with them, and the limited instruction set itself. A few exceptions or special instructions are fine, many are too much. The Z80 to me, seems to have just the right mix of regular instructions and limited special cases.

Hardware Problems

I have been working with IBM PS/2s lately and been finding out some of their problems. Went to upgrade my 50Z and found out the hard way about memory SIPs. Seems some early modules used a SIP holder design that breaks far to easily. Mine did just that. Only repair option is to send the boards back and get a new one with a different designed holder. Right now I have some plastic holding the memory SIP in place. A good temporary fix is getting a round piece of plastic and cutting it in half on one side only. This allows you to slip it over the ends (the cut part) where it will work as a mini "C" clamp and hold the SIP in place. If I had been a regular computer user and not one with lots of hardware experience I would have though the problem was something I did. But I handled the SIP insertion properly and still the holding tabs broke off. Also I can look at the newer boards and see they no longer use this style holder. That is definitely the sign that the original part was at fault. However good luck at trying to get IBM to admit they used faulty parts.

Next Time

Well that is about it for now. Hopefully I can get to some old ideas I want to cover next time. Until then keep working on hacking up those small projects. •

Companies Mentioned:

Video Dimensions
UltiMeth Systems
24035 Fernlake Drive
Harbor City, CA 90710
(213) 539-4276

Avocet Systems Inc.
P.O. Box 490
Rockport, Maine 04856
(800) 448-8500

Circuit Cellars INK
4 Park Street, Suite 20
Vernon, CT 06066
(203) 875-2751

The Computer Corner

by Bill Kibler

It seems like fatherhood sure has made a crimp in my computer work. Went to a swap meet yesterday and saw several items to buy, but decided I wouldn't have the time to work on them. A lot of real bargains to be had for those who want to play. I saw $8000 worth of S-100 system (new prices that is) go for $80. The chips alone in the system are worth more than that. It was a full hard disk with lots of extra interface cards.

VGA

Speaking of interface cards, my VGA is still keeping me going on the learning curve. Been having a real problem finding the type of information I want to see. My problem is a fixed frequency monitor (640 by 480 color) and software that likes to go in and change the VGA's modes. I had thought that once the VGA card was set for a certain mode type, it would convert calls to other modes into a compatible operation.

It turns out that any program, if given the chance, will change the mode of the card, even if you can't use that mode. The programs do not care about your setup, they think everybody is running multisync monitors (which explains why most have had to do so). I came across a program however that may solve my problems. It is called Video Dimensions by UltiMeth Systems. I am sure there are others like it, but this one was given to me to check out at work. What it appears to do is install a device driver for the VGA card. It also modifies the INT 10 jump address (INT 10 does the video operations in DOS—like set mode) to go to the device driver. It comes with some extra programs, different fonts, a VGA ANSI driver, and a utility program to set (or see if you need to set) all the possible VGA variables.

I like their V option, which puts up a color test pattern on the monitor. You can check your convergence to see if the monitor needs adjustment. You also have several modes for setting and checking the colors used. I was quickly able to see that mine needed the Horizontal control touched up some. What impressed me the most was being able to run some programs (like FPC) in their color CGA mode. Before using their driver, I got a non color display or it drove my monitor sync circuit out of control. Now it just works like it is suppose to. Time has not permitted me to test everything, but their book explains how to use it for a number of programs (WS with 35 lines).

What I am beginning to understand is that the better VGA card companies will supply new drivers for their VGA products. Those who don't (like mine) will give you drivers for special programs which will work if you use a multisync monitor. If you want to use a fixed frequency monitor and have other modes converted or mapped to the only mode you can use, a special device driver is needed. Video Dimensions appears to be a good device drive for those without multisync monitors.

LANS

My current work project is getting our old 68000 software to work on a LAN system. We have used dedicated serial lines for each terminal. Now days the terminal card sits in a PC, but we do not use the PC bus for data transfers, we still use serial ports. Our clients want to use LANS to speed up data throughput. Also they want to use regular LAN services as well as our special boards. I feel the clients needs are justified for the demand, however their expectations maybe a little over optimistic.

One fact that is little looked at in comparing speed of dedicated lines to LANS is sharing of resources. Typically, limits on a dedicated line are only the speed of the line and how fast the primary system can feed data to it. Ours run at 38.4KB with most of their time spent packetizing the data (memory limits had a 240 byte packet size) or waiting on the host for data. The LAN system can theoretically give higher throughput. In reality they seldom if ever achieve that throughput. When crashes with other packets, retries, and sometimes multiple packetizing of the data is taken into account, transfer rates can be pretty low. A fellow worker has tested packet size to transfer rates and found 2K packet sizes can produce rates as low as 25KB for a system sold as 10MB transfer rates.

My feelings are starting to lean toward another sales job. By that I mean, the specs would indicate considerably higher transfer rates, when in fact consistent rates are far below what those idealized design specs would indicate. I would therefore not believe anybody who is trying to sell you a system based ONLY on how much faster it will transfer your data. The features that you get from having a hard disk server system and common data files however is another matter, one not related to transfer speed.

A separate point is quoting a report by Infonetics of Santa Clara about LAN crashes. It seems they interviewed 100 big companies who use LANS. They reported an average of 23 system crashes a year. Each crash lasted about 5 hours. The cost of those crashes was about $26,000 each or $5,200 per hour of down time. The survey didn't indicate causes, but ours seems to go down often without reason. Some people feel power quality is a problem, I slant toward poor hardware and software design. My old S100 systems never failed because they didn't use switching power supplies. If you want a solid hardware LAN system, make sure it has an old style linear power supply. The LAN software is most likely similar to any major software product in which not all the bugs will ever be found.

Macros, We Have More and More Macros...

My major task at work, or I should say long term project, is porting the old 68000 assembly code from an Exormacs (Motorola development) system to be crossassembled on PC based system. I am using the Avocet crossassembler as it is suppose to be 100% compatible with the latest Motorola assembler we were using. The first problem I ran into was definition of com-

(Continued on page 39)